Cheminformatics in R

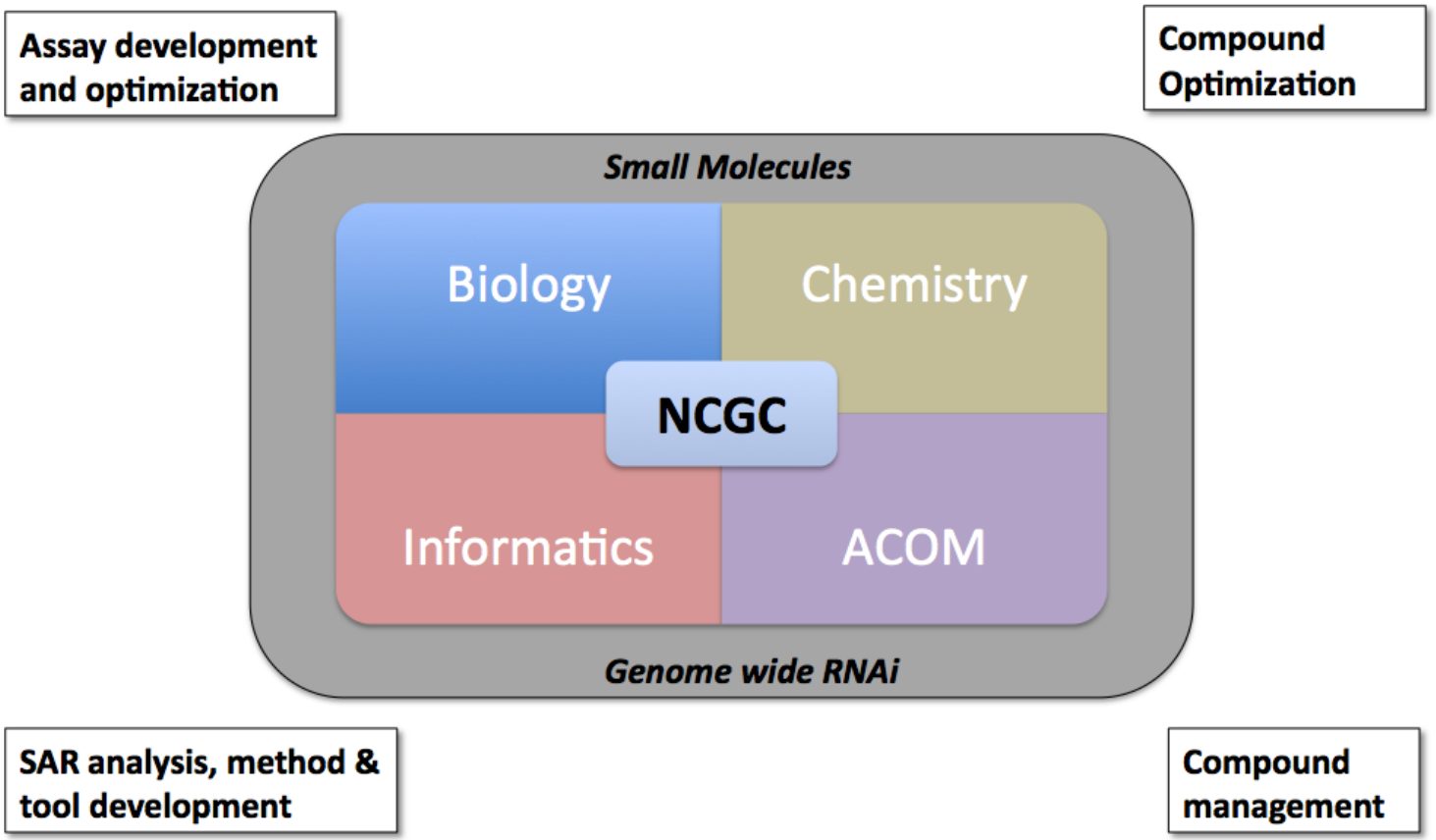
or *How to Start R and Never Have to Exit*

Rajarshi Guha

NIH Chemical Genomics Center

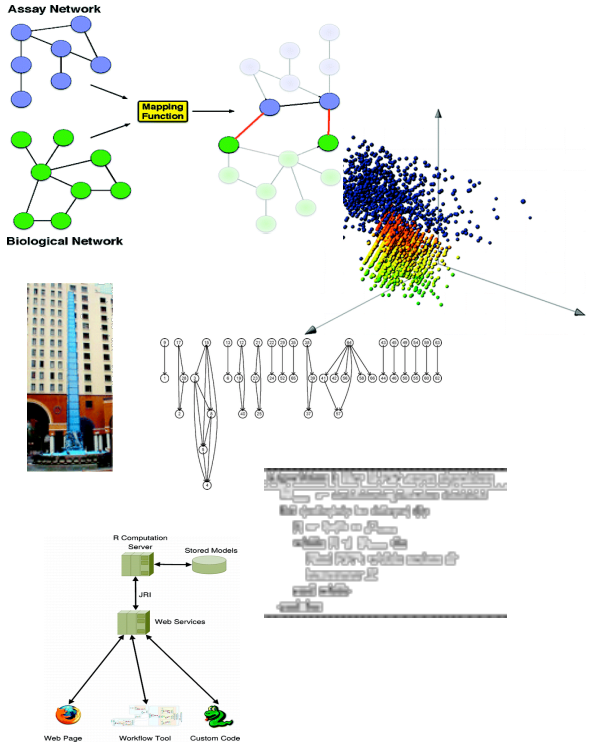
17th May, 2010
EBI, Hinxton

Background

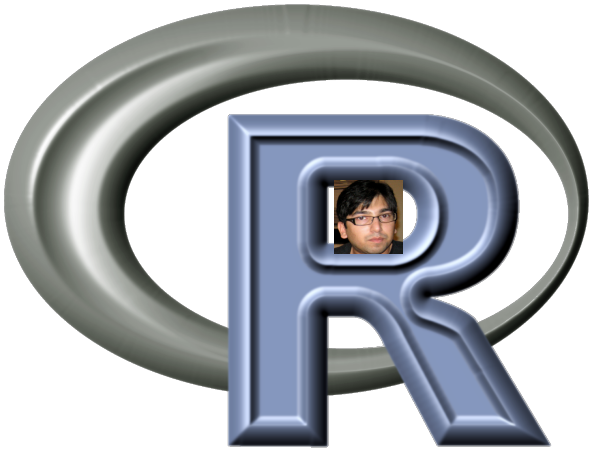

Assay development and optimization
Compound Optimization
Small Molecules
Biology
Chemistry
NCGC
Informatics
ACOM
Genome wide RNAi
SAR analysis, method & tool development
Compound management

Background

- Involved in various aspects of cheminformatics since 2002
  - QSAR modeling, virtual screening, polypharmacology, networks
  - Algorithm developement
  - Cheminformatics software development
- Core developer for the CDK



Background



- Pretty much live inside of R
- Been using it since 2003, developed a number of R packages, mostly public
- Make extensive use of R at NCGC for small molecule & RNAi screening

Acknowledgements

- `rcdk`
  - Miguel Rojas
  - Ranke Johannes
- CDK
  - Egon Willighagen
  - Christoph Steinbeck
  - ...

Resources

- Download binaries
  - If you're working on Unix it's a good idea to compile from source
- There's lots of documentation of varying quality
  - The R manual is a good document to start with
  - R Data Import and Export is **extremely** handy regarding data loading
  - A variety of short introductory tutorials as well as documents on specific topics in statistical modeling can be found here
- More general help can be obtained from
  - The R-help mailing list. Has a lot of renoowned experts on the list and if you can learn quite a bit of statistics in addition to getting R help just be reading the archives of the list. Note: they do expect that you have done your homework! See the posting guide
  - A useful reference for Python/Matlab users learning R

R Books

- A number of books are available ranging from introductions to R along with examples up to advanced statistical modeling using R.
- Data Analysis and Graphics Using R: An Example-based Approach by John Maindonald, John Braun
- Introductory Statistics with R by Peter Dalgaard
- Using R for Introductory Statistics by John Verzani
- If you end up programming a lot in R, you'll want to have Programming with Data: A Guide to the S Language by John M. Chambers

IDE's and editors

- Emacs + ESS - works on Linux, Windows, OS X. *Invaluable* if you are an Emacs user
- TINN-R is a small and useful editor for Windows
- You can also do R in Eclipse
  - StatET
  - Rsubmit
- A number of different GUI's are available for, which may be useful if you're an infrequent user
  - Rcommander
  - JGR
  - Rattle
  - PMG
  - SciViews-R
  - Red-R

What is R?

- R is an environment for modeling
  - Contains many prepackaged statistical and mathematical functions
  - No need to implement anything
- R is a matrix programming language that is good for statistical computing
  - Full fledged, interpreted language
  - Well integrated with statistical functionality
  - More details later

What is R?

- It is possible to use R just for modeling
- Avoids programming, preferably use a GUI
  - Load data → build model → plot data
- But you can also get much more creative
  - Scripts to process multiple data files
  - Ensemble modeling using different types of models
  - Implement brand new algorithms
- R is good for prototyping algorithms
  - Interpreted, so immediate results
  - Good support for vectorization
    - Faster than explicit loops
    - Analogous to `map` in Python and Lisp
  - Most times, interpreted R is fine, but you can easily integrate C code

What is R?

- R integrates with other languages
  - C code can be linked to R and C can also call R functions
  - Java code can be called from R and vice versa. See various packages at rosuda.org
  - Python can be used in R and vice versa using Rpy
- R has excellent support for publication quality graphics
- See R Graph Gallery for an idea of the graphing capabilities
- But graphing in R does have a learning curve
- A variety of graphs can be generated
  - 2D plots - scatter, bar, pie, box, violin, parallel coordinate
  - 3D plots - OpenGL support is available

Why Cheminformatics in R?

- In contrast to bioinformatics (cf. Bioconductor), not a whole lot of cheminformatics support for R
- For cheminformatics and chemistry relevant packages include
  - `rcdk`, `rpubchem`, `fingerprint`
  - `bio3d`, `ChemmineR`
- A lot of cheminformatics employs various forms of statistics and machine learning - R is exactly the environment for that
- We just need to add some chemistry capabilities to it

Outline

- An overview of the R language
- An overview of the CDK library
- Exploring the `rcdk` package
- Exploring the `rcdklibs` package
- Extending the code

Accessing packages

- From within R (you'll need 2.11.0 or better)

```
install.packages(c("rcdk","rpubchem"), dependencies = TRUE)
```

- Sources for `rcdk` and `rcdklibs` can be obtained from the Github repository
  - This is required if you want to modify Java code associated with `rcdk`
- Installable development packages (OS X, Linux, Windows) available from http://rguha.net/rcdk
  - These will make their way to CRAN

Workshop prerequisites

- Everything should be installed
- Running the code below should not give any errors

```
library(rcdk)
library(rpubchem)
source(helper.R)
```

- Most of the code snippets in these slides are contained in `code.R`
- You should have a `data/` directory containing various datasets used in this workshop
- You should have a `exercises/` directory containing the source code solutions to the exercises

Part I

Overview of R

Outline

The language

Parallel R

Database access

Working in R

- To get help for a function name do: `?functionname`
- If you don't know the name of the function, use: `help.search("blah")`
- R Site Search and Rseek are very helpful online resources
- To exit from the prompt do: `quit()`
- Two ways to run R code
  - Type or paste it at the prompt
  - Execute code from a source file: `source("mycode.R")`
- Saving your work
  - `save.image(file="mywork.Rda")` saves the whole workspace to *mywork.Rda*, which is a binary file
  - When you restart R, do: `load("mywork.Rda")` will restore your workspace
  - You can save individual (or multiple) objects using `save`

Basic types in R

Primitive types

- **numeric** - indicates an integer or floating point number
- **character** - an alphanumeric symbol (string)
- **logical** - boolean value. Possible values are TRUE and FALSE

Complex types

- **vector** - a 1D collection of objects of the same type
- **list** - a 1D container that can contain arbitrary objects. Each element can have a name associated with it
- **matrix** - a 2D data structure containing objects of the same type
- **data.frame** - a generalization of the matrix type. Each column can be of a different type

Primitive types

```
> x <- 1.2
> x <- 2

> y <- "abcdegf"
> y
[1] "abcdegf"

> z <- c(1,2,3,4,5)
> z
[1] 1 2 3 4 5

> z <- 1:5
> z
[1] 1 2 3 4 5
```

Matrices

- Make a matrix from a vector
- The matrix is constructed column-wise

```
> z <- c(1,2,3,4,5,6,7,8,9,10)
> m <- matrix(z, nrow=2, ncol=5)
> m
     [,1] [,2] [,3] [,4] [,5]
[1,]  1  3  5  7  9
[2,]  2  4  6  8  10
```

data.frame

- We want to store compound names and number of atoms and whether they are toxic or not
- Need to store: characters, numbers and boolean values

```
x <- c("aspirin", "potassium cyanide",
       "penicillin", "sodium hydroxide")
y <- c(21, 3, 41,3)
z <- c(FALSE, TRUE, FALSE, TRUE)
d <- data.frame(x,y,z)

> d
                x y z
1 aspirin 21 FALSE
2 potassium cyanide 3 TRUE
3 penicillin 41 FALSE
4 sodium hydroxide 3 TRUE
```

data.frame

- But do the column means? 6 months later, we probably won't know (easily)
- So we should add names

```
> names(d) <- c("Name", "NumAtom", "Toxic")
> d
              Name NumAtom Toxic
1 aspirin 21 FALSE
2 potassium cyanide 3 TRUE
3 penicillin 41 FALSE
4 sodium hydroxide 3 TRUE
```

- When adding column names, don't use the underscore character
- You can access a column of a `data.frame` by it's name: `d$Toxic`

Lists

- A 1D collection of arbitrary objects (`ArrayList` in Java)
- We can access the lists by indexing: `mylist[1]` or `mylist[[1]]` for the first element or `mylist[c(1,2,3)]` for the first 3 elements

```
x <- 1.0
y <- "hello there"
s <- c(1,2,3)
mylist <- list(x,y,s)

> mylist
[[1]]
[1] 1

[[2]]
[1] "hello there"

[[3]]
[1] 1 2 3
```

Lists

- It's useful to name individual elements of a list

```
x <- "oxygen"
z <- 8
m <- 32
mylist <- list(name="oxygen",
               atomicNumber=z,
               molWeight=m)
> mylist
$name
[1] "oxygen"

$atomicNumber
[1] 8

$molWeight
[1] 32
```

Lists

- We can then get the molecular weight by writing

```
> mylist$molWeight
[1] 32
```

Identifying types

- It's useful initially, to identify the type of the object
- Usually just printing it out explains what it is
- You can be more concise by doing

```
> x <- 1
> class(x)
[1] "numeric"

> x <- "hello world"
> class(x)
[1] "character"

> x <- matrix(c(1,2,3,4), nrow=2)
> class(x)
[1] "matrix"
```

Indexing

- Indexing fundamental to using R and is applicable to vectors, matrices, data.frame's and lists
- all indices start from 1 (not 0!)
- For a 1D vector, we get the i'th element by `x[i]`
- For a 2D matrix of data.frame we get the i, j element by `x[i,j]`
- But we can also index using vectors
  - Called an *index vector*
  - Allows us to select or exclude multiple elements simultaneously

Indexing

- Say we have a vector, `x` and a matrix `y`

```
> x
 [1] 1 2 3 4 5 6 7 8 9 10
> y
     [,1] [,2] [,3] [,4] [,5]
[1,]  1  4  7  10  13
[2,]  2  5  8  11  14
[3,]  3  6  9  12  15
```

- Some ways to get elements from `x`
- 5th element $\longrightarrow$ `x[5]`
- The 3rd, 4th and 5th elements $\longrightarrow$ `x[ c(3,4,5) ]`
- Everything **but** the 3rd, 4th and 5th elements $\longrightarrow$ `x[ -c(3,4,5) ]`

Indexing

- Say we have a matrix `y`

```
> y
     [,1] [,2] [,3] [,4] [,5]
[1,]  1  4  7  10  13
[2,]  2  5  8  11  14
[3,]  3  6  9  12  15
```

- Some ways to get elements from `y`
- Element in the first row, second column → `y[1,2]`
- All elements in the 1st column → `y[,1]`
- All elements in the 3rd row → `y[3,]`
- Elements in the first 2 columns → `y[ , c(1,2) ]`
- Elements **not** in the first 2 columns → `y[ , -c(1,2) ]`

Indexing

- You can also use a boolean vector to index another vector
- In this case, the index vector must of the same length as your target vector
- If the i'th element in the index vector is TRUE then the i'th element of the target is selected

```
> x <- c(1,2,3,4)
> idx <- c(TRUE, FALSE, FALSE, TRUE)
> x[ idx ]
[1] 1 4
```

- It's a pain to have to create a whole index vector by hand

Indexing

- A more intuitive way to do this, is to make use of R's vector operations
- `x == 2` will compare each element of `x` with the value 2 and return `TRUE` or `FALSE`
- The result is a boolean vector

```
> x <- c(1,2,3,4)
> x == 2
[1] FALSE TRUE FALSE FALSE
```

- So we can select the elements of x that are equal to 2 by doing

```
> x <- c(1,2,3,4)
> x[ x == 2 ]
[1] 2
```

Indexing

- Or select elements of x that are greater than 2

```
> x <- c(1,2,3,4)
> x[ x > 2 ]
[1] 3 4
```

- Or select elements of x that are 2 *or* 4

```
> x <- c(1,2,3,4)
> x[ x == 2 | x == 4 ]
[1] 2 4
```

- The single bar (|) is intentional
- Logical operations using vectors should use |, & rather than the usual ||, &&

List indexing

- With list objects we can index using `[` or `[[`
  - `[` keeps element names, `[[` drops names
- Can't use index vectors or boolean vectors with `[[`

```
> y <- list(a='b', b=123, x=c(1,2,3))
> y[3]
$x
[1] 1 2 3

> y[[3]]
[1] 1 2 3

> y[[1:2]]
Error in y[[1:2]] : subscript out of bounds
```

Subsetting

- Get a subset of the rows of a matrix or `data.frame` based on values in a certain column
- We can generate a boolean index vector
- For `data.frame`'s, we can use a more intuitive approach

```
d <- data.frame(a=1:10, b=runif(10), c=rnorm(10))

d[d$a <= 6,]

subset(d, a <= 6)
subset(d, a >= 7 & b < 0.4)
```

Merging data.frames

- Lets you join data.frames based on a common column
- Pretty much identical to an SQL join (and supports natural and outer joins)
- Very handy when merging data from multiple sources

```
d1 <- data.frame(mol=I(c("mol1", "mol2", "mol3")),
                 klass=c("active", "active", "inactive"))
d2 <- data.frame(molecule=I(c("mol1", "mol2", "mol3")),
                 TPSA=c(1.2,3.4,5.6), Kier1=c(5.7, 8.9, 12))

> merge(d1, d2, by.x="mol", by.y="molecule")
   mol klass TPSA Kier1
1 mol1 active 1.2 5.7
2 mol2 active 3.4 8.9
3 mol3 inactive 5.6 12.0
```

Functions in R

- Functions are much like in any other language
- R employs copy-on-write semantics
  - Send in a copy of an input variable if it will be modified in the function
  - Otherwise send in a reference
- Typing is dynamic

```
my.add <- function(x,y) {
  return(x+y)
}
```

- You can then call the function as `my.add(1,2)`
- In general, check for the type of object using `class()`
- Functions can be anonymous

Loop constructs

- R does not have C-style looping

```
for (i = 0; i < 10; i++) {
  print(i)
}
```

- Rather, it works like Python's for-in idiom

```
for (i in c(0,2,3,4,5,6,7,8,9)) {
  print(i)
}
```

Loop constructs

- In general, to loop n times, you create a vector with n elements, say by writing $1 : n$
- But like Python loops, you can just loop over elements of a vector or list and use them

```
> objects <- list(x=1, y="hello world", z=c(1,2,3))
> for (i in objects) print(i)
[1] 1
[1] "hello world"
[1] 1 2 3
```

- Good R style discourages explicit loops

Apply and friends

- Explicit loops (`for (...)`) are not idiomatic and can be slow
- `apply`, `lapply`, `sapply` resemble functional programming styles (but see `Reduce`, `Filter` and `Map` for alternatives)
- Good idea to master these forms

```
x <- c(1,2,3,4,5)
sapply(x, sqrt)
sapply(x, function(z) z+3)

sapply(x, function(z) rep(z,3), simplify=FALSE)
sapply(x, function(z) rep(z,3), simplify=TRUE)
```

- Handy for looping over vector elements

Looping over lists

- For list objects, use `lapply`
- The return value is a list - if it can be converted to a simple vector use `unlist` to do so

```
x <- list(x=1, b=2, c=3)

lapply(x, sqrt)
unlist(lapply(x, sqrt))

lapply(x, function(z) {
  return(z^2)
})
```

- The function argument of `lapply` should expect an object as the first argument
- Columns of a `data.frame` are lists, so `lapply` can be used to loop over columns

Looping over multiple objects

- What if you want to loop over two (or more) lists (or vectors) simulataneously
  - Like Pythons `zip` would allow
- Use `mapply` and provide a function that takes (at least) as many arguments as input vectors/lists

```
x <- 1:3
y <- list(a="Hello", b=12.34, c=c(1,2,3))
mapply(function(a,b) {
  print(a)
  print(b)
  print("---")
}, x, y)
```

Looping over matrices

- Use `apply` to operate on entire rows or columns of a matrix or `data.frame`
- Be careful when working on rows of a `data.frame` - behavior can be unexpected if all columns are not of the same type

```
m <- matrix(1:12, nrow=4)
apply(m, 1, sum) ## sum the rows
apply(m, 2, sum) ## sum the columns
```

- The function argument of `apply` should expect a vector as the first argument

Looping caveats

- The main thing to remember when using `apply`, `lapply` etc is nature of inputs to the user supplied function
- `apply` will send a vector (corresponding to a row or column)
- `lapply` and `sapply` will send a single element of the list/vector
- `mapply` will send a single element of each of the input lists/vectors
- The function argument of these methods can take anonymous functions or pre-defined functions

Loading data

- Writing vectors and matrices by hand is good for 15 minutes!
- Easier to read data from files
- R supports a variety of text and binary formats
- Certain packages can be loaded to handle special formats
  - Read from SPSS, Stata, SAS
  - Read microarray data, GIS data, chemical structure data
- Excel files can also be read (easiest on Windows)
- Data can be accessed from SQL databases (Postgres, MySQL, Oracle)

Reading CSV data

- Consider the extract of a CSV file below

```
ID,Class,Desc1,Desc2,Desc3,Desc4,Desc5
w150,nontoxic,0.37,0.44,0.86,0.44,0.77
c49,toxic,0.37,0.58,0.05,0.85,0.57
s97,toxic,0.66,0.63,0.93,0.69,0.08
```

- Key features include
  - A header line, naming the columns
  - An ID column and a Class column, both characters
- Reading this data file is as simple as

```
> dat <- read.csv("data1.csv", header=TRUE)
```

- If you print `dat` it will scroll down your screen

Reading CSV data

- To get a quick summary of the `data.frame` use the `str()` function

```
> str(dat)
'data.frame': 100 obs. of 7 variables:
 $ ID : Factor w/ 100 levels "a115","a180",..: 83 7 66 85 64 82 33 10 92 9
 $ Class: Factor w/ 2 levels "nontoxic","toxic": 1 2 2 1 2 1 1 1 1 2 ...
 $ Desc1: num 0.37 0.37 0.66 0.18 0.88 0.37 0.72 0.72 0.2 0.68 ...
 $ Desc2: num 0.44 0.58 0.63 0.09 0.1 0.14 0.03 0.15 0.65 0.66 ...
 $ Desc3: num 0.86 0.05 0.93 0.57 0.85 0.67 0.99 0.68 0.77 0.78 ...
 $ Desc4: num 0.44 0.85 0.69 0.85 0.58 0.17 0.95 0.19 0.69 0.61 ...
 $ Desc5: num 0.77 0.57 0.08 0.17 0.74 0.56 0.22 0.19 0.91 0.13 ...
```

What is a factor?

- The `factor` type is used to represent categorical variables
  - Toxic, non-toxic
  - Active, inactive
  - Yes, no, maybe
- They correspond to enum's in C/Java
- A factor variable looks like an ordinary vector of characters
- Internally they are represented by integers
- A factor variable has a property called the levels
  - Indicates what values are valid for the factor

```
> levels(dat$Class)
[1] "nontoxic" "toxic"
```

What is a factor?

- If you try to assign an invalid value to a factor you get a warning

```
> dat$Class[1] <- "Yes"
Warning message:
In `[<-.factor`(`*tmp*`, 1, value = "Yes") :
  invalid factor level, NAs generated

> dat$Class[1:10]
 [1] <NA> toxic toxic nontoxic toxic nontoxic nontoxic nontoxic
 [9] nontoxic toxic
Levels: nontoxic toxic
```

Operating on groups

- Factors are very useful when you'd like to operate on subsets of the data, as defined by factor levels
- Assay data might label molecules as active, inactive, inconclusive
- `by` is similar to `lapply` but operates on `data.frame`'s in a factor-wise way
- Takes a function argument which will receive a subset of the input `data.frame`, corresponding to a single level of the factor

```r
assay <- read.csv("data/aid2170.csv", header=TRUE)
str(assay)
levels(assay$PUBCHEM.ACTIVITY.OUTCOME)

## how many actives and inactives are there?
by(assay, assay$PUBCHEM.ACTIVITY.OUTCOME, function(x) {
  nrow(x)
})

## summary stats of the inhibiion values, by group
by(assay, assay$PUBCHEM.ACTIVITY.OUTCOME, function(x) {
  summary(x$Inhibition)
})

## or distributions of the inhibition values, by group
par(mfrow=c(1,2))
by(assay, assay$PUBCHEM.ACTIVITY.OUTCOME, function(x) {
  hist(x$Inhibition)
})
```

Outline

The language

Parallel R

Database access

Parallel R

- R itself is not multi-threaded
  - Well suited for embarassingly parallel problems
- Even then, a number of "large data" problems are not tractable
  - Recent developments on integrating R and Hadoop address this
  - See the `RHIPE` package
- We'll look at `snow` which allows distribution of processing on the same machine (multiple CPU's) or multiple machines
- But see `snowfall` for a nice set of wrappers around `snow`
- Also see `multicore` for a package that focuses on parallel processing on multicore CPU's

Trivial parallelization with `snow`

- `snow` lets you use multiple machines or multiple cores on a single machine
- Well suited for data-parallel problems
- If using multiple machines, data will be sent across the network, so large chunks of data can slow things down
- General strategy is
  - Initialize cluster
  - Send variables required for calculation to the nodes
  - Call a parallel function

Use case - exhaustive feature selection

- Given a set of N descriptors, find a n-descriptor subset that leads to a good predictive model
- Will obviously explode - ${}^{50}C_5 = 2,118,760$ descriptor combinations
- Usually we would employ a genetic algorithm or simulated annealing or even some form of stepwise selection
- For pedagogical purposes
  - how can we do this exhaustively?
  - how can we speed it up?

The data and a model

- Lets consider some melting point data
- Precalculated and reduced descriptor set
- 184 molecules in training set, 34 descriptors
- *Goal - find a good 4 descriptor OLS model*

[0]Bergstrom, C.A.S. et al, *J. Chem. Inf. Model.*, **2003**, 43, 1177–1185

A serial solution

- Evaluate all 4-member combinations of the numbers $\{1, 2, \ldots, 34\}$
  - These are the indices of the descriptor columns
- Loop over the rows, use the values as column indices of the descriptor matrix, build a model

```
## First some data prep
library(gtools)
load("data/bergstrom/bergstrom.Rda")

depv <- train.desc[,1]
desc <- train.desc[,-1]

idx <- 1:nrow(desc)
combos <- combinations(ncol(desc), 4)
```

A serial solution

```
rmse <- function(x,y) sqrt(sum((x-y)^2)/length(x))

system.time(
  rmse.serial <- apply(combos, 1, function(idx, mydata, y) {
    dat <- data.frame(y=y, x=mydata[,idx])
    m <- lm(y~., dat)
    return(rmse(m$fitted,y))
  }, mydata=desc, y=depv)
)
```

- Takes about 200 seconds

The parallel solution

- Code is pretty much identical
- First need to initialize the "cluster" - in this case we'll run multiple processes on the same machine

```
library(snow)
clus <- makeCluster(rep("localhost",2), type="SOCK")

system.time(
  rmse.par <- parApply(clus, combos, 1, function(idx, mydata, y) {
    dat <- data.frame(y=y, x=mydata[,idx])
    m <- lm(y~., dat)
    return(rmse(m$fitted,y))
  }, mydata=desc, y=depv)
)
```

- Takes about 120 seconds, but easily scales up with multiple cores

Outline

The language

Parallel R

Database access

R and databases

- Bindings to a variety of databases are available
  - Mainly RDBMS's but some NoSQL databases are being interfaced
- The R `DBI` spec lets you write code that is portable over databases
- Note that loading multiple database packages can lead to problems
- This can happen even when you don't explicitly load a database package
  - Some Bioconductor packages load the `RSQLite` package as a dependency, which can interfere with, say, `ROracle`

A quick look at RSQLite

- Not suitable for very large data or multiple connections
- Definitely look at sqldf, that makes it very easy to use SQL on R `data.frames`
- Very brief overview of what we can do with `RSQLite` for local storage
- Should transer easily to other databases

Basic usage - connecting

- We'll use a database I prepared and placed at `data/db/assay.db`
- Lets first get some information about the tables and field definitions

```
library(RSQLite)

## specific to RSQLite
sqlite <- dbDriver("SQLite")
con <- dbConnect(sqlite, dbname = "data/db/assay.db")

> dbListTables(con)
[1] "assay"

> dbListFields(con, "assay")
[1] "row_names" "aid" "PUBCHEM_SID" "PUBCHEM_CID"
  "PUBCHEM_ACTIVITY_OUTCOME" "PUBCHEM_ACTIVITY_SCORE"
  "PUBCHEM_ASSAYDATA_COMMENT"
```

Basic usage - querying

- The simplest way to get data is to slurp in the whole table as a `data.frame`
- Not a good idea if the table is very large

```
assay <- dbReadTable(con, "assay")

> str(assay)
'data.frame': 188264 obs. of 6 variables:
 $ aid : num 396 396 396 429 429 429 429 429 429 429 ...
 $ PUBCHEM_SID : int 10318951 10318962 10319004 842121 842122 842123 84212
 $ PUBCHEM_CID : int 6419748 5280343 969516 6603008 6602571 6602616 644371
 $ PUBCHEM_ACTIVITY_OUTCOME : chr "active" "active" "active" "inactive" ..
 $ PUBCHEM_ACTIVITY_SCORE : int 100 100 100 0 1 1 1 -3 0 0 ...
 $ PUBCHEM_ASSAYDATA_COMMENT: int 20061024 20060421 20061024 20061127 2006
```

Basic usage - querying

- Better to get subsets of the table via SQL queries
- Two ways to do it - get back all rows from the query or retrieve in chunks
- The latter is better when you might get a lot of rows

```
## query and get results in one go
dbGetQuery(con, "select distinct aid from assay")

## query and then get results in chunks
res <- dbSendQuery(con, "select * from assay where aid = 429")
fetch(res, n=10) ## get first 10 results
fetch(res, n=10) ## get next 10
fetch(res, n=-1) ## get remaining results

## no more rows to fetch
fetch(res, n=-1)

dbClearResult(res) ## a vital step!
```

Saving data in a database

- Can directly write a `data.frame` to a database file
- RSQLite will automatically generate a `CREATE TABLE` statement

```
library(rpubchem)

## get some assay data from PubChem
assay <- sapply(c(396, 429, 438, 445), function(x) {
  data.frame(aid=x, get.assay(x))[,1:6]
}, simplify=FALSE)

## join all assays into a single data.frame
assay <- do.call("rbind", assay)

## Make a new table
sqlite <- dbDriver("SQLite")
con <- dbConnect(sqlite, dbname = "assay.db")
dbWriteTable(con, "assay", assay)
dbDisconnect(con)
```

So why use a database?

- Dont have to load bulk CSV or .Rda files each time we start work
- Can index data in RDBMS's so queries can be very fast
- Good way to exchange data between applications (as opposed to .Rda files which are only useful between R users)
- All the code above, except for the `dbConnect` statement will be identical for PostgreSQL, MySQL etc.

Part II

The Chemistry Development Kit

Outline

Resources

- The CDK wiki
- The nightly build page
- Code snippets
- cdk-user mailing list
- Keyword list and feature list
- If you want to develop *with* the CDK, use the comprehensive jar file that contains all dependencies
  - from the nightly build page for the cutting edge
  - or from Sourceforge for the last stable release

Some CDK issues

- Learning the CDK is non-trivial
- No real tutorial but see here for a start
- Best place is to use the keyword list to find the relevant class and then look at the unit tests for a class

What does the CDK provide?

- Fundamental chemical objects
  - atoms
  - bonds
  - molecules
- More complex objects are also available
  - Sequences
  - Reactions
  - Collections of molecules
- Input/Output for a wide variety of molecular file formats
- Fingerprints and fragment generation
- Rigid alignments, pharmacophore searching
- Substructure searching, SMARTS support
- Molecular descriptors

Some design principles

- Abstraction is the key principle
  - Applies to chemical objects such as atoms, bonds
  - Also applied to input/output
- Rather than directly creating an atom object we use a factory method

```
IAtom atom = DefaultChemObjectBuilder.getInstance().
               newAtom();
```

and not

```
  IAtom atom = new Atom();
```

Some design principles

- By using `DefaultChemObjectBuilder` we don't worry how an atom or bond is created
- By using this type of abstraction we can load any file format without having to specify it
- Another aspect is the use of interfaces rather than concrete types
- A molecule is a collection of atoms and bonds
  - A graph view is not imposed directly
  - Molecular features such as aromaticity are properties of the atoms and bonds

Molecules

- A molecule is fundamentally represented as an IAtomContainer object
- For many purposes this is fine
- In some cases specialization is required so we have
  - Crystal
  - Ring
  - Fragment
- Some methods require a specific subclass
- Many methods simply require an IAtomContainer, so you can use any of the above subclasses

Molecules

- Similar procedure to getting an atom or bond object

```
IAtomContainer container = DefaultChemObjectBuilder.
                           getInstance().
                           newAtomContainer();
```

- Then we populate it

```
container.addAtom( atom1 );
container.addAtom( atom2 );
container.addBond( atom1, atom2, 1.5);
```

Outline

Introduction

I/O

Fingerprints

Substructures

Input/Output

- SMILES are a common format
- SMILES parsing, you will have to handle the `InvalidSmilesException`

```
SmilesParser sp = new SmilesParser(
                DefaultChemObjectBuilder.
                 getInstance());
IMolecule molecule = sp.parseSmiles("CCCC(CC)CC=CC=CC");
```

- Given a molecule object, we can create a SMILES

```
SmilesGenerator sg = new SmilesGenerator();
String smiles = sg.createSMILES(molecule);
System.out.println("SMILES = "+smiles);
```

Input/Output

- Reading files from disk is a common task
- A little more complex due to abstraction but is quite general

```
File file = new File("mols.sdf");
FileReader fileReader = new FileReader(file);
MDLV2000Reader reader = new MDLV2000Reader(fileReader);
IChemFile chemFile = (IChemFile) reader.
                     read(new ChemFile());
List containers = ChemFileManipulator.
                  getAllAtomContainers(chemFile);
```

Input/Output

- This is nice since it allows you to get all the molecules in the file at one go
- Not a good idea for very large SD files
- In such cases, use IteratingMDLReader
- Allows you to iterate over arbitrarily large SD files
- There is also an IteratingSMILESReader for big SMILES files

Input/Output

- Writing files is also supported for a wide variety of formats
- For example, to write to SD format

```
FileWriter w1 = new FileWriter(new File("molecule.sdf"));
try {
   MDLWriter mw = new MDLWriter(w1);
   mw.write(molecule);
   mw.close();
} catch (Exception e) {
   System.out.println(e.toString());
}
```

SD tags

- SD tags are a common way to associate property information with a molecular structure
- See PubChem SD files to get an idea of what we can put in them

```
  CDK 1/28/07,15:24

  2 1 0 0 0 0 0 0 0 0999 V2000
    0.0000 0.0000 0.0000 C 0 0 0 0 0 0 0 0 0 0 0 0
    0.0000 0.0000 0.0000 C 0 0 0 0 0 0 0 0 0 0 0 0
  1 2 1 0 0 0 0
M END
> <myProperty>
1.2

> <anotherProperty>
Hello world
```

SD tags

- When a molecule is read in from an SD file we can get the value of a given tag by doing

```
Object value = molecule.getProperty("myProperty");
```

- When writing a molecule we can supply a set of tags in a HashMap

```
HashMap tags = new HashMap();
tags.put("myProperty", new Double(1.2));
```

SD tags

- To ensure tags are written to disk we would do

```
FileWriter w1 = new FileWriter(new File("molecule.sdf"));
try {
   MDLWriter mw = new MDLWriter(w1);
   mw.setSdFields( tags );
   mw.write(molecule);
   mw.close();
} catch (Exception e) {
   System.out.println(e.toString());
}
```

SD tags

- A molecule might have several properties associated with it.
- We can avoid creating an extra HashMap, when writing them *all* to disk

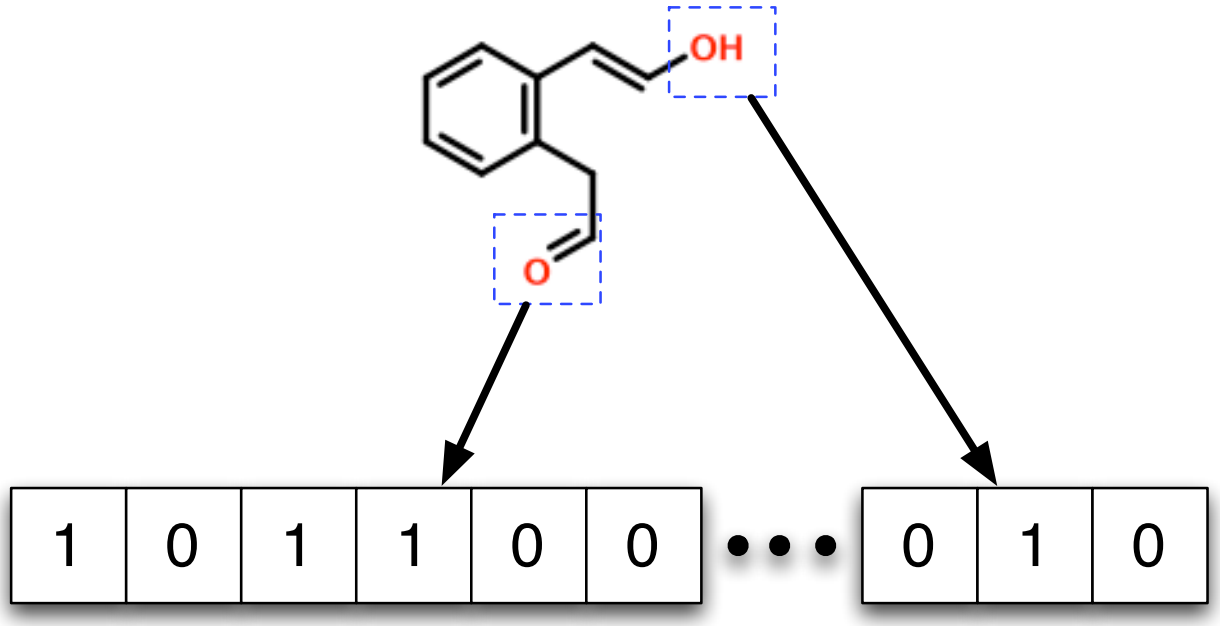
```
MDLWriter mw = new MDLWriter(w1);
mw.setSdFields( molecule.getProperties() );
mw.write(molecule);
mw.close();
```

Generating canonical SMILES

- The SmilesGenerator class will create canonical SMILES

```
String smiles = "C(C)(C)CC=CC(C(CC(C))CC)CC";
SmilesParser sp = new SmilesParser();
IMolecule mol = sp.parseSmiles(smiles);

SmilesGenerator sg = new SmilesGenerator();
String canSmi = sg.createSMILES(mol);
```

Fingerprints



- Used in many areas - database screening, diversity analysis, modeling

Outline

Fingerprints

- The simplest fingerprint looks for specific substructures and sets a specific bit
  - MACCSFingerprinter
  - SubstructureFingerprinter
- Hashed fingerprints don't maintain a correspondence between bit position and substructural feature

Getting fingerprints

- The CDK can generate binary fingerprints
- This gives a default fingerprint of 1024 bits - but you can specify smaller or larger fingerprints
- The CDK also provides variants of the default fingerprint
  - MACCSFingerprint
  - ExtendedFingerPrinter - includes bits related to rings
  - GraphOnlyFingerprinter - simplified version of fingerprinter that ignores bond order
  - SubstructureFingerprinter - generates fingerprints based on a specified set of substructures

```
IFingerprinter fprinter = new Fingerprinter()

BitSet fingerprint = fprinter.getFingerprint(molecule);
```

Evaluating similarity

- Given two molecules we are interested in determining how similar they are
- A common approach is to evaluate their fingerprints
- Calculate a similarity metric (e.g., Tanimoto coefficient)

```
BitSet fp1 = Fingerprinter.getFingerprint(molecule1);
BitSet fp2 = Fingerprinter.getFingerprint(molecule2);

float tc = Tanimoto.calculate(fp1, fp2);
```

Outline

Substructure searching

- Identify the presence/absence of a substructure in a molecule

```
IAtomContainer mol = ...;

SMARTSQueryTool sqt = new SMARTSQueryTool("C");
sqt.setSmarts("C(=O)C");
boolean matched = sqt.matches(mol);
```

- In many cases we're interested in all the possible matches

```
List<List<Integer>> maps;
maps = sqt.getUniqueMatchingAtoms();
```

- `maps.size()` gives you the number of matches
- Each element of `maps` is a sequence of atom ID's

Pharmacophores

- We'll look at constructing a pharmacophore query by hand
- Generally, you'd read it from a file
- Given a query we can search for that in a target molecule
- The target molecule should have 3D coordinates
- Generally you'll examine multiple conformers for a given molecule

Pharmacophores

- Construct pharmacophore groups

```
PharmacophoreQueryAtom o =
   new PharmacophoreQueryAtom("D", "[!H0;#7,#8,#9]");
PharmacophoreQueryAtom n1 =
   new PharmacophoreQueryAtom("A", "c1ccccc1");
PharmacophoreQueryAtom n2 =
   new PharmacophoreQueryAtom("A", "c1ccccc1");
```

Pharmacophores

- Construct pharmacophore constraints

```
PharmacophoreQueryBond b1 =
  new PharmacophoreQueryBond(o, n1, 4.0, 4.5);
PharmacophoreQueryBond b2 =
   new PharmacophoreQueryBond(o, n2, 4.0, 5.0);
PharmacophoreQueryBond b3 =
   new PharmacophoreQueryBond(n1, n2, 5.4, 5.8);
```

Pharmacophores

- Given pharmacophore groups and constraints, we construct a pharmacophore query
- Design to have the same semantics as a normal molecule

```
QueryAtomContainer query =
    new QueryAtomContainer();

query.addAtom(o);
query.addAtom(n1);
query.addAtom(n2);

query.addBond(b1);
query.addBond(b2);
query.addBond(b3);
```

Pharmacophores

- Given the query we can now perform a pharmacophore search

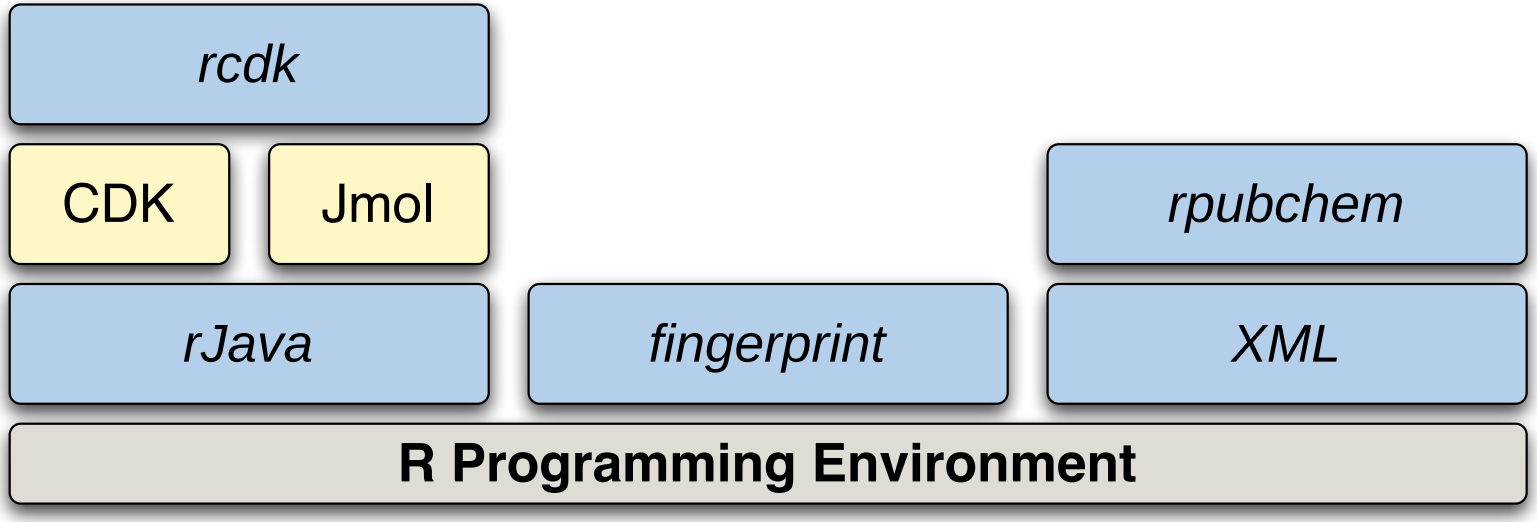
```
IAtomContainer aMolecule;

// load in the molecule

PharmacophoreMatcher matcher =
    new PharmacophoreMatcher(query);
boolean status = matcher.matches(aMolecule);
if (status) {
  // get the matching pharmacophore groups
  List<List<PharmacophoreAtom>> pmatches =
     matcher.getUniqueMatchingPharmacophoreAtoms();
}
```

Part III

Cheminformatics & R - `rcdk`

Using the CDK in R

- Based on the `rJava` package
- Two R packages to install (not counting the dependencies)
- Provides access to a variety of CDK classes and methods
- Idiomatic R



Outline

I/O

Molecular data

Visualization

Descriptors

Fingerprints

Fragments

Reading in data

- The CDK supports a variety of file formats
- `rcdk` loads all recognized formats, automatically
- Data can be local or remote

```
mols <- load.molecules( c("data/io/set1.sdf",
                          "data/io/set2.smi",
                          "http://rguha.net/rcdk/remote.sdf"))
```

- Gives you a `list` of Java references representing `IAtomContainer` objects
- Can't do much with these objects, except via `rcdk` functions

Reading structures from text files

- SMILES strings can be contained in arbitrary text files (such as CSV)
- If you read such a file with `read.table` or `read.csv`, make sure to set `comment.char=""`
- Also a good idea to specify the `colClasses` argument or else `as.is=TRUE`- otherwise SMILES will be read in as factors

Reading SMILES strings

- Also useful to load molecules directly from a SMILES string
- `parse.smiles` parses a single molecule

```
mol <- parse.smiles("c1ccccc1C(=O)N")

## loop over multiple SMILES
smiles <- c("CCC", "c1ccccc1",
            "C(C)(C=O)C(CCNC)C1CC1C(=O)")
mols <- sapply(smiles, parse.smiles)
```

- But the second call is inefficient, especially for large SMILES collections

Efficient SMILES parsing

- By default *parse.smiles* instantiates a SMILES parser object (`SmilesParser`)
- So when looping over a vector of SMILES strings we're creating a parser object each time
- In such a case, specify a parser object explicitly - 2X speedup

```
smilesParser <- get.smiles.parser()

## now specify this parser
smiles <- c("CCC","c1ccccc1","C(C)(C=O)C(CCNC)C1CC1C(=O)")
mols <- sapply(smiles, parse.smiles, parser = smilesParser)
```

Efficient SMILES parsing


```
> parser <- get.smiles.parser()
> smiles <- rep(c("CCC",
                  "c1ccccc1",
                  "C(C)(C=O)C(CCNC)C1CC1C(=O)"),
                 500)

> system.time(junk <- sapply(smiles, parse.smiles))
   user system elapsed
  4.088 0.106 3.946

> system.time(junk <- sapply(smiles, parse.smiles, parser=parser))
   user system elapsed
  1.792 0.032 1.705
```

You get what is provided . . . and no more

- CDK philosophy is not to do extra work
- For file reading this implies that it only parses what is specified in the file and will not perform operations that are not explicitly indicated
- As a result some features (isotopic masses) of molecules and atoms are not automatically configured
- `load.molecule` **will do these extra steps**
- `parse.smiles` **will not**

Configuring molecules


```
## no explicit configuration needed
mols <- load.molecules("data/io/bp.smi")
get.exact.mass(mols[[1]])

## explicit configuration required
mol <- parse.smiles("c1ccccc1")
get.exact.mass(mol)
```

- The last call will give a `NullPointerException` since the isotopic masses have not been configured

Configuring molecules

- When parsing SMILES, do the configuration by hand

```
smi <- "OC(=O)C1=C(C=CC=C1)OC(=O)C"
mol <- parse.smiles(smi)

## do the configuration
do.typing(mol) ## not required in recent versions
do.aromaticity(mol) ## not required in recent versions
do.isotopes(mol)

## now, this works
get.exact.mass(mol)
```

Hydrogens - implicit & explicit

- SMILES indicates that when hydrogens are not specified, they are to be considered implicit
- But MDL MOL files have no such specification, so if no H's specified, the molecule will not have explicit or implicit hydrogens

```
mol <- load.molecules("data/noh.mol")[[1]]
unlist(lapply(get.atoms(mol), get.symbol))
```

- In such cases, `convert.implicit.to.explicit` will add implicit H's (to satisfy valencies) and then convert them to explicit

```
convert.implicit.to.explicit(mol)
unlist(lapply(get.atoms(mol), get.symbol))
```

A note on function calls

- R uses copy-on-write semantics for function calls
  - If a function modifies its input argument, R will send a copy of the input variable to the function
  - Otherwise send a reference
  - "Safety" of call-by-value, efficiency of call-by-reference
- In contrast, `convert.implicit.to.explicit` modifies the input argument
  - No return value
- In general `rJava` allows you to work with Java objects using call-by-reference semantics

Writing molecules

- Currently only SDF is supported as an output file format
- By default a multi-molecule SDF will be written
- Properties are not written out as SD tags by default

```
smis <- c("c1ccccc1", "CC(C=O)NCC", "CCCC")
mols <- sapply(smis, parse.smiles)

## all molecules in a single file
write.molecules(mols, filename="mols.sdf")

## ensure molecule data is written out
write.molecules(mols, filename="mols.sdf", write.props=TRUE)

## molecules in individual files
write.molecules(mols, filename="mols.sdf", together=FALSE)
```

Generating SMILES

- Finally, we can also create SMILES strings from molecules via `get.smiles`
- Works on a single molecule at a time
- Canonical SMILES are generated by default

```
smis <- c("c1ccccc1", "CC(C=O)NCC", "CCCC")
mols <- sapply(smis, parse.smiles)

lapply(mols, get.smiles)
```

Outline

I/O

Molecular data

Visualization

Descriptors

Fingerprints

Fragments

Accessing attached data

- Some file formats allow you to attach data to a structure
- SDF is a good example, using named tags and associated values
- For example, a DrugBank SDF has fields such as

```
> <DRUGBANK_ID>
DB00135

> <DRUGBANK_GENERIC_NAME>
L-Tyrosine

> <DRUGBANK_MOLECULAR_FORMULA>
C9H11NO3

> <DRUGBANK_MOLECULAR_WEIGHT>
181.1885
```

Accessing data fields

- The CDK reads SDF tags and their values into the `IAtomContainer` object
- We can access them via the `get.property` and `get.properties`

```
> mols <- load.molecules("data/io/set1.sdf")
> get.properties(mols[[1]])
$DRUGBANK_ID
[1] "DB00135"

$DRUGBANK_IUPAC_NAME
[1] "(2S)-2-amino-3-(4-hydroxyphenyl)propanoic acid"

$`cdk:Title`
[1] "135"

...
```

Accessing data fields

- If you know the name of the field, just pull the value directly

```
get.property(mols[[1]], "DRUGBANK_GENERIC_NAME")
```

- Note that values for **all** tags are `character`, so you need to cast them to the appropriate types manually

Extra tags

- In some cases, you'll see tag names that are not in the SDF file
- The CDK uses molecule property fields to attach arbitrary data.
- For example, the title field from a SDF entry is stored using the `cdk:Title` name
- In general, all CDK-derived properties have names prefixed by "`cdk:`"

```
## identify CDK-derived properties
props <- get.properties(mols[[1]])
props[ grep("^cdk:", names(props)) ]
```

Setting property values

- You can add your own property values
- The type of the value is preserved, as long as you are within R
- The type of the name (or key) must be `character`

```
mol <- parse.smiles("c1ccccc1Cc2ccccc2")
set.property(mol, "MyProperty", 23.14)
set.property(mol, "toxic", "yes")
get.properties(mol)
```

Persisting molecular data

- You could extract the properties and write them to a text file

```
mols <- load.molecules("data/io/set1.sdf")
props <- lapply(mols, get.properties)

## convert list to table and write it out
props <- do.call("rbind", props)
write.table(props, "drugdata.txt", row.names=FALSE)
```

Persisting molecular data

- Molecular data can also be persisted by writing to SDF format

```
mol <- parse.smiles("c1ccccc1")
set.property(mol, "foo", 1)
set.property(mol, "bar", "baz")
write.molecules(mol, "mol.sdf", write.props=TRUE)
```

Working with molecules

- Currently you can access atoms, bonds, get certain atom properties, 2D/3D coordinates
- Since `rcdk` doesn't cover the entire CDK API, you might need to drop down to the `rJava` level and make calls to the Java code by hand
- I'm happy to code specific tasks in idiomatic R

Working with molecules

- A useful task is to *wash* or *standardize* molecules
- CDK doesn't provide a method to do this directly
- But a common operation is to check for disconnected molecules and keep the largest fragment

```
> m <- parse.smiles("c1ccccc1")
> is.connected(m)
[1] TRUE
>
> m <- parse.smiles("C1CC(N=C1)C(=O)[O-].[Na+]")
> is.connected(m)
[1] FALSE
>
> largest <- get.largest.component(m)
> length(get.atoms(largest))
[1] 8
```

Getting atoms & bonds

- Simple to get atoms and bonds

```
mol <- parse.smiles("c1ccccc1C(Cl)(Br)c1ccccc1")

atoms <- get.atoms(mol)
bonds <- get.bonds(mol)
```

- These are lists of `jobjRef` objects
- `rcdk` currently supports getting the symbol, coordinates, atomic number, implicit H-count
- Can also check whether an atom is aromatic etc.
- `?get.symbol`

Working with atoms

- Simple elemental analysis
- Identifying flat molecules

```
mol <- parse.smiles("c1ccccc1C(Cl)(Br)c1ccccc1")
atoms <- get.atoms(mol)

## elemental analysis
syms <- unlist(lapply(atoms, get.symbol))
round( table(syms)/sum(table(syms)) * 100, 2)

## is the molecule flat?
coords <- do.call("rbind", lapply(atoms, get.point3d))
any(apply(coords, 2, function(x) length(unique(x)) == 1))
```

SMARTS matching

- `rcdk` supports substructure searches with SMARTS or SMILES
- May not be practical for large collections of molecules due to memory

```
mols <- sapply(c("CC(C)(C)C",
                 "c1ccc(Cl)cc1C(=O)O",
                 "CCC(N)(N)CC"), parse.smiles)
query <- "[#6D2]"
hits <- matches(query, mols)

> print(hits)
CC(C)(C)C c1ccc(Cl)cc1C(=O)O CCC(N)(N)CC
FALSE TRUE TRUE
```

Persisting objects

- We already saw how we can persist molecule properties
- It'd be useful to also persist CDK objects (i.e., `jobjRef`'s)

```
m <- parse.smiles("c1ccccc1")

obj <- .jserialize(m)

newObj <- .junserialize(obj)
```

- Serialized objects are just `raw` vectors, so can be saved via `save`
- But also take a look at `.jcache`

Outline

I/O

Molecular data

Visualization

Descriptors

Fingerprints

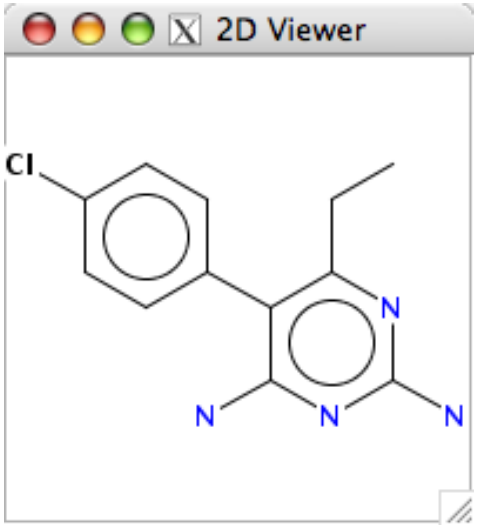
Fragments

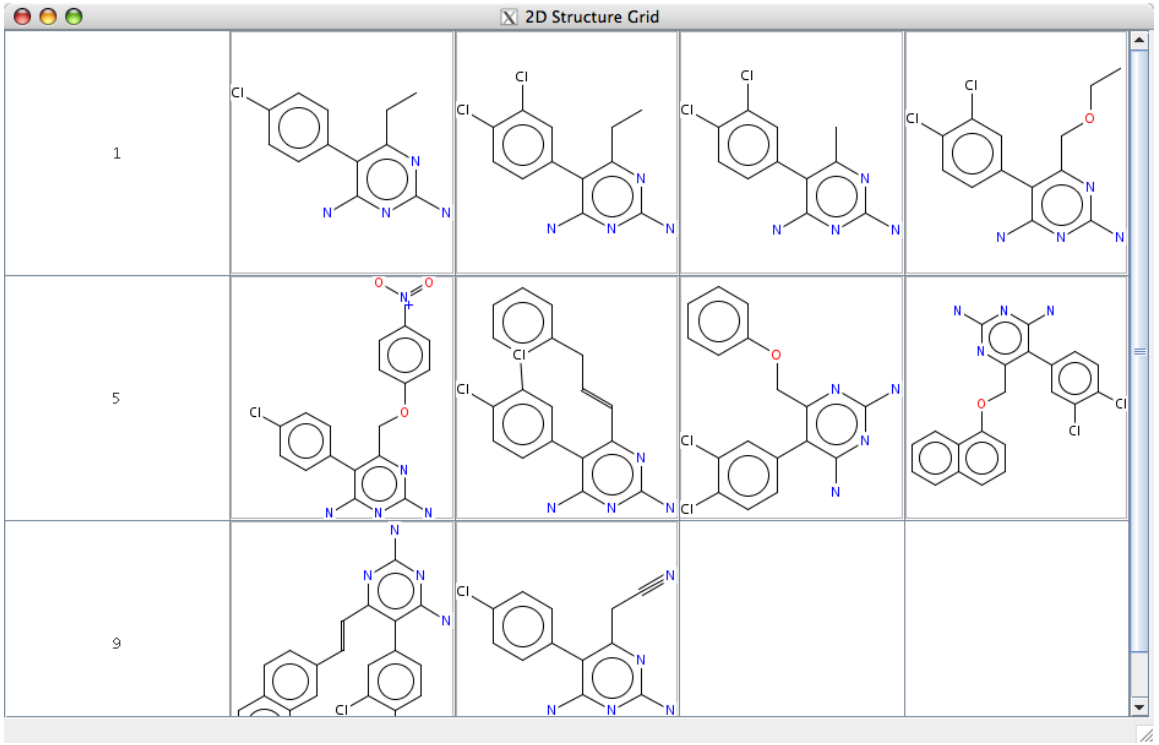
Visualization

- `rcdk` supports visualization of 2D structure images in two ways
- First, you can bring up a Swing window
- Second, you can obtain the depiction as a raster image
- **Doesn't work on OS X**

```
mols <- load.molecules("data/dhfr_3d.sd")

## view a single molecule in a Swing window
view.molecule.2d(mols[[1]])

## view a table of molecules
view.molecule.2d(mols[1:10])
```

Visualization





Visualization

- The Swing window is a little heavy weight
- It'd be handy to be able to annotate plots with structures
- Or even just make a panel of images that could be saved to a PNG file
- We can make use of `rasterImage` and `rcdk`
- As with the Swing window, this won't work on OS X

Visualization

```
m <- parse.smiles("c1ccccc1C(=O)NC")
img <- view.image.2d(m, 200,200)

## start a plot
plot(1:10, 1:10, pch=19)

## overlay the structure
rasterImage(img, 1,8, 3,10)
```

Visualization

- By playing around with plotting parameters we can fill up the plot region with an image
- The values being plotted define the coordinates used to overlay the raster image

```
mols <- load.molecules("data/dhfr_3d.sd")

## A table of 16 structures
par(mfrow=c(4,4), mar=c(0,0,0,0))

for (i in 1:16) {
  img <- view.image.2d(mols[[i]], 200, 200)
  plot(1:10, xaxt="n", yaxt="n", xaxs="i", yaxs="i", col="white")
  rasterImage(img, 1,1, 10,10)
}
```

Outline

I/O

Molecular data

Visualization

Descriptors

Fingerprints

Fragments

Molecular descriptors

- Numerical representations of chemical structure features
- Can be based on
  - connectivity
  - 3D coordnates
  - electronic properties
  - combination of the above
- *Many* descriptors are described and implemented in various forms
- The CDK implements 45 descriptor classes, resulting in $\approx$ 300 individual descriptor values for a given molecule

Descriptor caveats

- Not all descriptors are optimized for speed
- Some of the topological descriptors employ graph isomorphism which makes them slow on large molecules
- In general, to ensure that we end up with a rectangular descriptor matrix we do not catch exceptions
- Instead descriptor calculation failures return `NA`

CDK Descriptor Classes

- The CDK provides 3 packages for descriptor calculations
  - `org.openscience.cdk.qsar.descriptors.molecular`
  - `org.openscience.cdk.qsar.descriptors.atomic`
  - `org.openscience.cdk.qsar.descriptors.bond`
- `rcdk` only supports molecular descriptors
- Each descriptor is also described by an ontology
  - For `rcdk` this is used to classify descriptors into groups

Descriptor calculations

- Can evaluate a single descriptor or all available descriptors
- If a descriptor cannot be calculated, `NA` is returned (so no exceptions thrown)
- First need to get available descriptor class names

```
dnames <- get.desc.names()
```

- Gives you a vector of all descriptor class names

Descriptor calculations

- You can also specify descriptor categories, to get a subset

```
dnames <- get.desc.names("topological")

## what categories are available?
> get.desc.categories()
[1] "electronic" "protein" "topological" "geometrical" "constitutional" "h
```

- Depending on the input structures, some descriptors may not make sense
- If you evaluate 3D descriptors from SMILES input, they'd all be set to `NA`

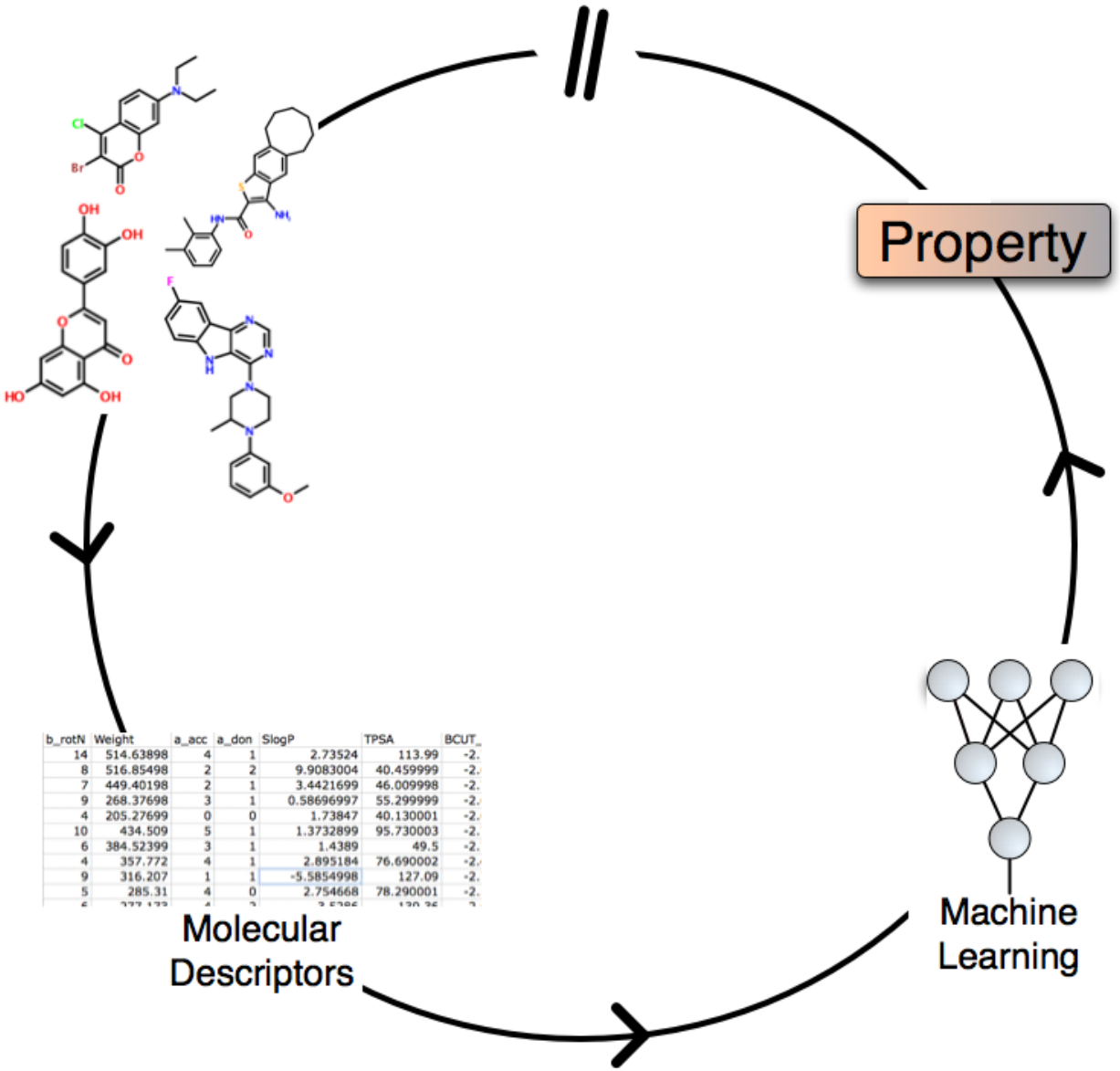
Evaluating the descriptors

- With a set of descriptor class names in hand, you can evaluate them

```
descs <- eval.desc(mols, dnames)
```

- `descs` will be a `data.frame` which can then be used in any of the modeling functions
- Column names are the descriptor names provided by the CDK
- Good practise to put molecule titles in a column or in the row names

The QSAR workflow



b_rotN	Weight	a_acc	a_don	SlogP	TPSA	BCUT_
14	514.63898	4	1	2.73524	113.99	-2.
8	516.85498	2	2	9.9083004	40.459999	-2.
7	449.40198	2	1	3.4421699	46.009998	-2.
9	268.37698	3	1	0.58696997	55.299999	-2.
4	205.27699	0	0	1.73847	40.130001	-2.
10	434.509	5	1	1.3732899	95.730003	-2.
6	384.52399	3	1	1.4389	49.5	-2.
4	357.772	4	1	2.895184	76.690002	-2.
9	316.207	1	1	-5.5854998	127.09	-2.
5	285.31	4	0	2.754668	78.290001	-2.

The QSAR workflow

- Before model development you'll need to clean the molecules, evaluate descriptors, generate subsets
- With the numeric data in hand, we can proceed to modeling
- Before building predictive models, we'd probably explore the dataset
  - Normality of the dependent variable
  - Correlations between descriptors and dependent variable
  - Similarity of subsets
- Then we can go wild and build all the models that R supports

Outline

Accessing fingerprints

- CDK provides several fingerprints
  - Path-based, MACCS, E-State, PubChem
- Access them via `get.fingerprint(...)`
- Works on one molecule at a time, use `lapply` to process a list of molecules
- This method works with the `fingerprint` package
  - Separate package to represent and manipulate fingerprint data from various sources (CDK, BCI, MOE)
  - Uses C to perform similarity calculations
  - Lots of similarity and dissimilarity metrics available

Accessing fingerprints


```
mols <- load.molecules("data/dhfr_3d.sd")

## get a single fingerprint
fp <- get.fingerprint(mols[[1]], type="maccs")

## process a list of molecules
fplist <- lapply(mols, get.fingerprint, type="maccs")
```

- Each fingerprint is an S4 object
- See the `fingerprint` package man pages for more details

Reading fingerprints

- We can also read in fingerprint data generated by other tools - BCI, MOE, CDK

```
fplist <- fp.read("data/fp/fp.data", size=1052,
                  lf=bci.lf, header=TRUE)
```

- Easy to support new line-oriented fingerprint formats by providing your own line parsing function
- See `bci.lf` for an example

Similarity metrics

- The `fingerprint` package implements 28 similarity and dissimilarity metrics
- All accessed via the `distance` function (so you call this even if you want similarity)
- Implemented in C, but still, large similarity matrix calculations are not a good idea!

```
## similarity between 2 individual fingerprints
distance(fplist[[1]], fplist[[2]], method="tanimoto")
distance(fplist[[1]], fplist[[2]], method="mt")

## similarity matrix - compare similarity distributions
m1 <- fp.sim.matrix(fplist, "tanimoto")
m2 <- fp.sim.matrix(fplist, "carbo")

par(mfrow=c(1,2))
hist(m1, xlim=c(0,1))
hist(m2, xlim=c(0,1))
```

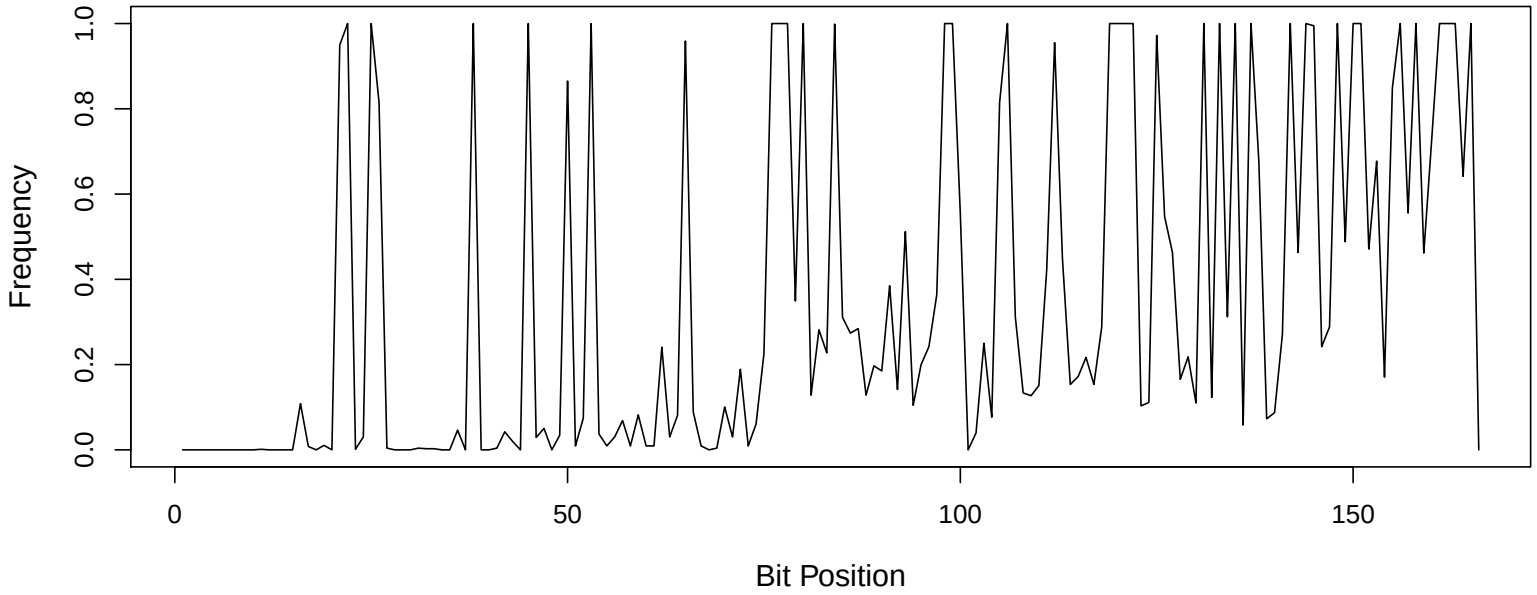
Comparing datasets with fingerprints

- We can compare datasets based on a fingerprints
- Rather than perform pairwise comparisons, we evaluate the normalized occurence of each bit, across the dataset
- Gives us a *n*-D vector - the "bit spectrum"

```
bitspec <- bit.spectrum(fplist)
plot(bitspec, type="l")
```

[0] Guha, R., *J. Comp. Aid. Molec. Des.*, **2008**, *22*, 367–384

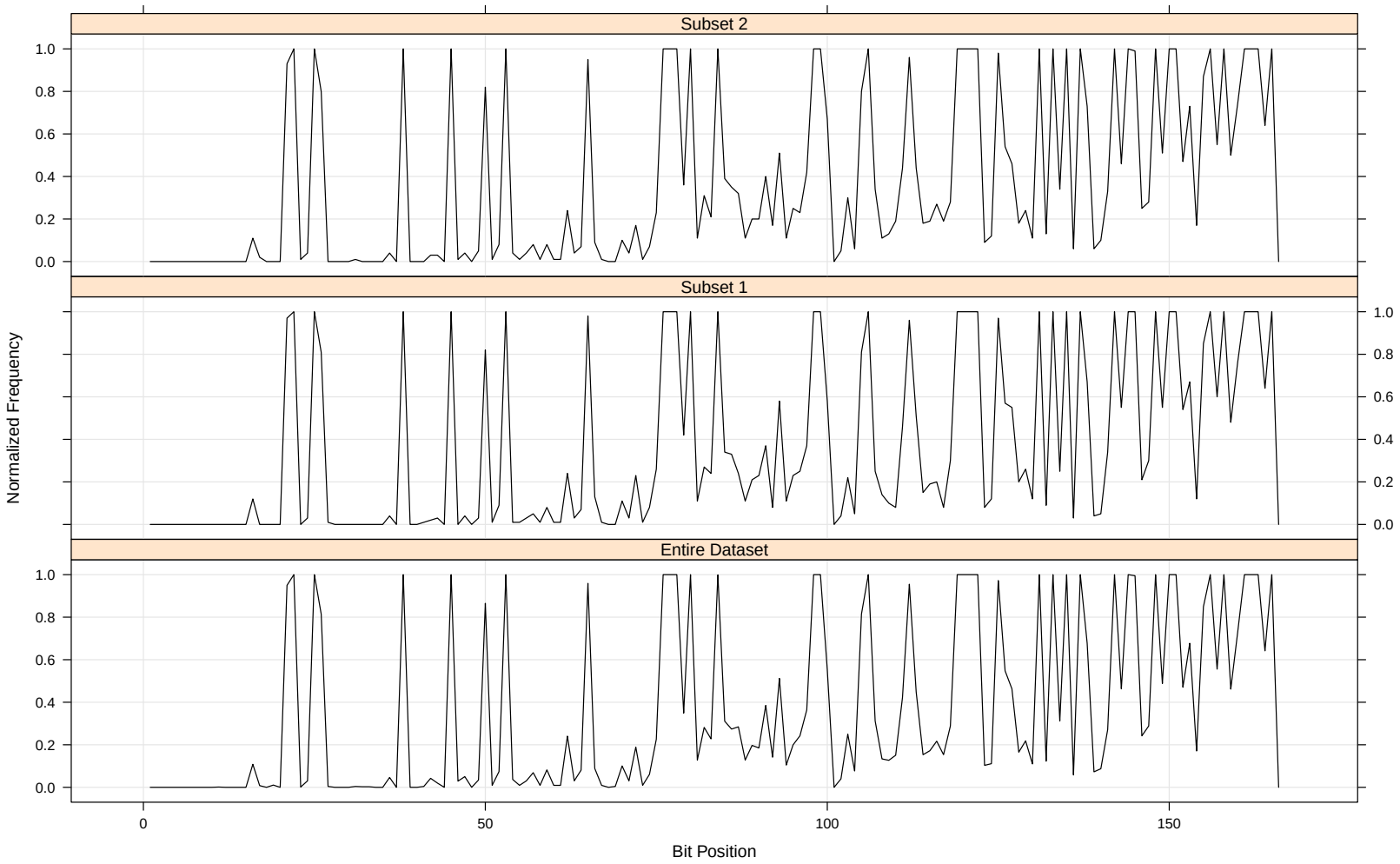
Bit spectrum



- Only makes sense with structural key type fingerprints
- Longer fingerprints give better resolution
- Comparing bit spectra, via any distance metric, allows us to compare datasets in $O(n)$ time, rather than $O(n^2)$ for a pairwise approach

[0] Guha, R., *J. Comp. Aid. Molec. Des.*, **2008**, *22*, 367–384

Comparing datasets

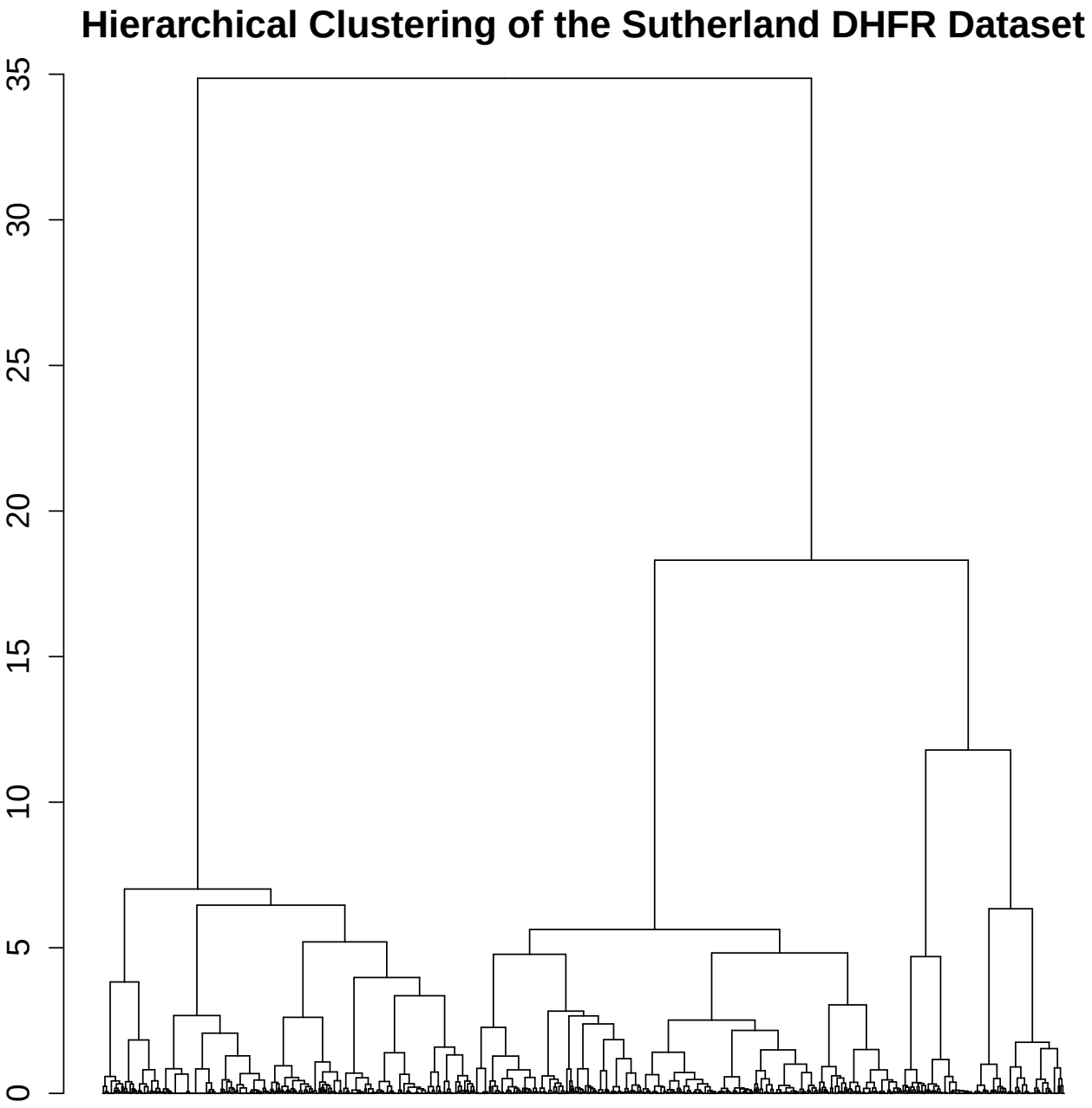


Fingerprints & clustering

- Clustering large collections is not practical
- Any way to avoid a pairwise similarity matrix is desirable

```
## Get fingerprints
mols <- load.molecules("data/dhfr_3d.sd")
fplist <- lapply(mols, get.fingerprint, type="maccs")

## Gives us a similarity matrix
fpdist <- fp.sim.matrix(fplist)

## Convert to a distance matrix
fpdist <- as.dist(1-fpdist)

## Perform the clustering
clus <- hclust(fpdist, method="ward")
```

Fingerprints & clustering

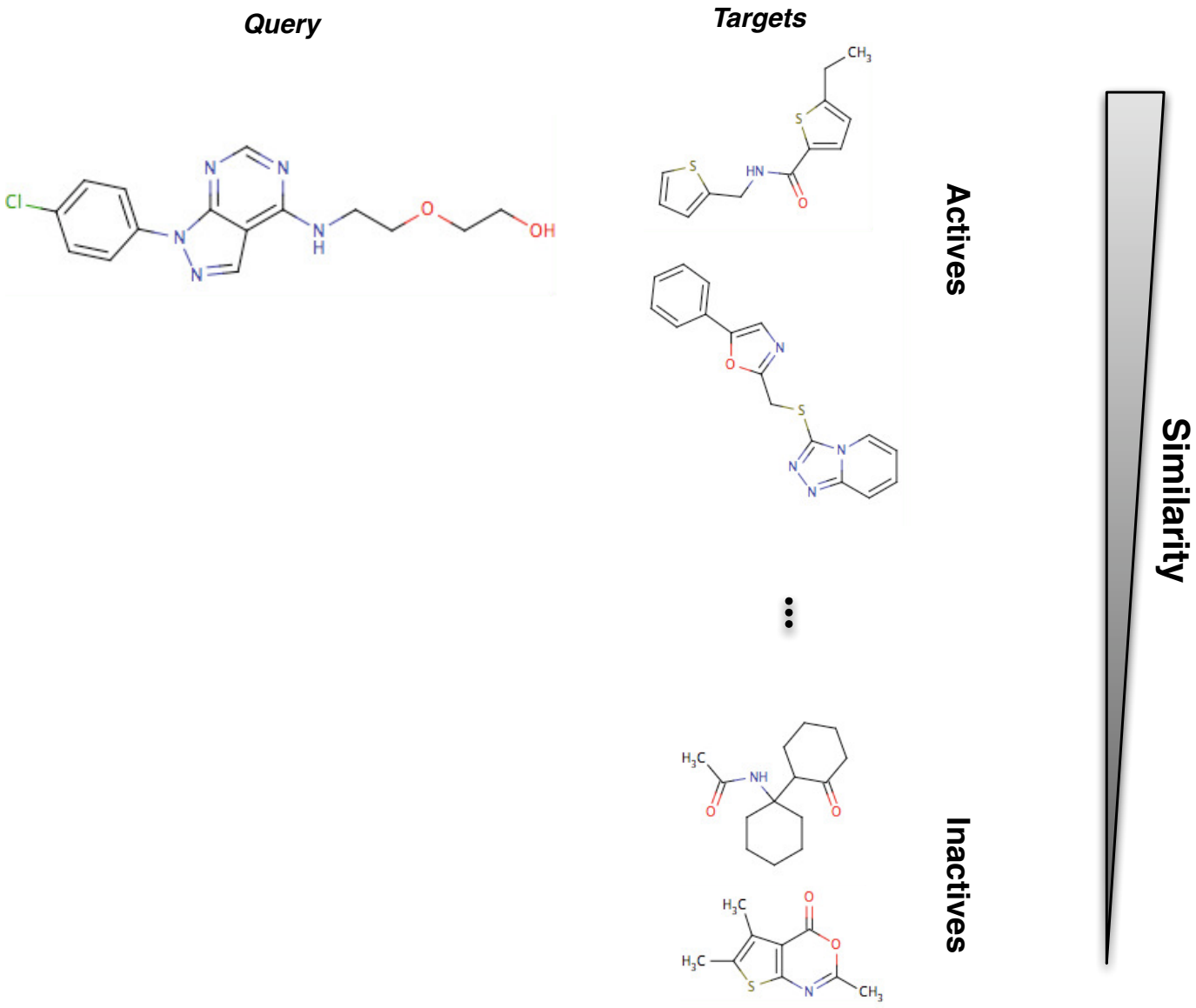


Comparing fingerprint performance

- Various studies comparing virtual screening methods
- Generally, metric of success is how many actives are retrieved in the top $n\%$ of the database
- Can be measured using ROC, enrichment factor, etc.
- Exercise - evaluate performance of CDK fingerprints using enrichment factors
  - Load active and decoy molecules
  - Evaluate fingerprints
  - For each active, evaluate similarity to all other molecules (active and inactive)
  - For each active, determine enrichment at a given percentage of the database screened

For a given query molecule, order dataset by decreasing similarity, look at the top 10% and determine fraction of actives in that top 10%

Comparing fingerprint performance

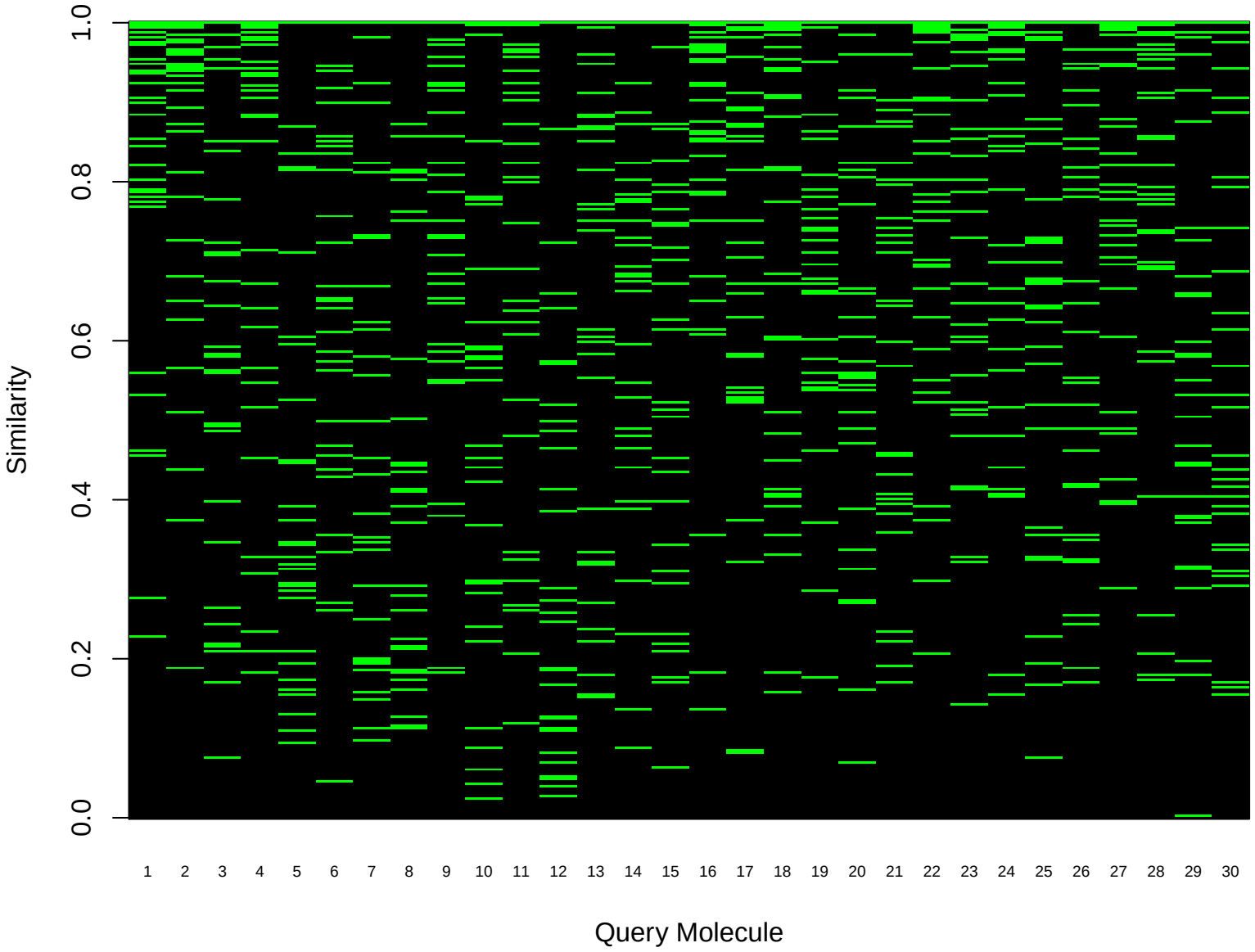


Comparing fingerprint performance

- A good dataset to test this out is the Maximum Unbiased Validation datasets by Rohr & Baumann
- Derived from 17 PubChem bioassay datasets and designed to avoid *analog bias* and *artifical enrichment*
- As a result, 2D fingerprints generally show poor performance on these datasets (by design)
- See here for a comparison of various fingerprints using two of these datasets

[0]Rohrer, S.G et al, *J. Chem. Inf. Model*, **2009**, *49*, 169–184
[0]Good, A. & Oprea, T., *J. Chem. Inf. Model*, **2008**, *22*, 169–178
[0]Verdonk, M.L. et al, *J. Chem. Inf. Model*, **2004**, *44*, 793–806

Comparing fingerprint performance



Outline

Fragment based analysis

- Fragment based analysis can be a useful alternative to clustering, especially for large datasets
- Useful for identifying interesting series
- Many fragmentation schemes are available
  - Exhaustive
  - Rings and ring assemblies
  - Murcko
- The CDK supports fragmentation (still needs work) into Murcko frameworks and ring systems
- We can also use the NCGC fragmenter (see `data/fragmenter.jar`) to exhaustively generate fragments

Getting fragments

- `rcdk` currently wraps the `GenerateFragments` class
- Only exposes the Murcko fragmentation method
- Lets look at directly working a CDK class and its methods
- Once we have the fragmenter we can perform a Murcko fragmentation and get back the fragments as SMILES

Getting fragments

```
mol <- parse.smiles(
"c1cc(c(cc1c2c(nc(nc2CC)N)N)[N+](=O)[O-])NCc3ccc(cc3)C(=O)N4CCCCC4")

## get the fragmenter
fragmenter <- .jnew("org/openscience/cdk/tools/GenerateFragments")

## do fragmentation
.jcall(fragmenter, "V", "generateMurckoFragments",
      .jcast(mol, "org/openscience/cdk/interfaces/IMolecule"),
      TRUE, TRUE, as.integer(6))

## get fragments as SMILES
frags <- .jcall(fragmenter,
              "[S",
              "getMurckoFrameworksAsSmileArray")
```

Some caveats

- The fragment SMILES come out as non-aromatic
- There is supposed to be a way to get the appropriate SMILES out, currently looking into this
- Implies that if we want to look at *fragment properties*, we're stuck
- However, we can still use these to look at series and/or local trends etc.

Aggregating fragments

- Lets wrap the fragmentation procedure into a function

```
get.frags <- function(mol, fragmenter) {
  .jcall(fragmenter, "V", "generateMurckoFragments",
         .jcast(mol, "org/openscience/cdk/interfaces/IMolecule"),
         TRUE, TRUE, as.integer(6))
  return(.jcall(fragmenter, "[S", "getMurckoFrameworksAsSmileArray"))
}
```

Aggregating fragments

- Now, we can get fragments for a set of molecules

```
mols <- load.molecules("data/dfhr_3d.sd")
fragmenter <- .jnew("org/openscience/cdk/tools/GenerateFragments")
frags <- lapply(mols, function(x, f) {
    get.frags(x, f)
}, f=fragmenter)
```

Aggregating fragments

- This gives us a list of fragment SMILES

```
> frags[1:3]
[[1]]
[1] "C1=CC=C(C=C1)C=2C=NC=NC=2"

[[2]]
[1] "C1=CC=C(C=C1)C=2C=NC=NC=2"

[[3]]
[1] "C1=CC=C(C=C1)C=2C=NC=NC=2"
```

- What we want is a `data.frame` that associates fragments, fragment ID's and the SMILES from which they are derived

Aggregating fragments

```
frags <- lapply(mols, function(x, f) {
  tmp <- get.frags(x, f)
  tmp <- data.frame(frag=I(tmp),
                    mol=I(get.property(x, "cdk:Title")))
  return(tmp)
}, f=fragmenter)
frags <- do.call("rbind", frags)
```

- Now we have something useful to work with

```
> head(frags)
                          frag mol
1 C1=CC=C(C=C1)C=2C=NC=NC=2 1-pyrimethamine
2 C1=CC=C(C=C1)C=2C=NC=NC=2 1-3062
3 C1=CC=C(C=C1)C=2C=NC=NC=2 1-7364
4 C1=CC=C(C=C1)C=2C=NC=NC=2 1-115194
5 C1=CC=C(C=C1)OCC=2C=CN=CN=2 1-118203
6 C1=CC=C(C=C1)C=2C=NC=NC=2 1-118203
```

Aggregating fragments

- Finally we generate some arbitrary fragment ID's

```
ufrags <- data.frame(frag=I(unique(frags$frag)),
                     fid=I(sprintf("F%03d")),
                         1:length(unique(frags$frag)))))
frags <- merge(frags, ufrags, by="frag")
```

- ... giving us

```
> tail(frags)
                                        frag mol fid
963 O=C2NC=NC=3NC=C(CNC1=CC=CC=C1)C2=3 43-13 F239
964 O=C2NC4=NC=NC=C4(C=3CN(CC1=CC=CC=C1)CC2=3) 49-4 F265
965 O=S(=O)(C1=CC=CC=C1)C=2C=CC3=NC=NC=C3(N=2) 22-8 F150
966 O=S(=O)(C1=CC=CC=C1)C=2C=CC3=NC=NC=C3(N=2) 22-9 F150
967 O=S(=O)(C1=CC=CC=C1)C=2C=CC3=NC=NC=C3(N=2) 22-7 F150
968 O=S(=O)(NCCOC1=CC=CC=C1)C2=CC=CC=C2 1-122059 F068
```

Doing stuff with fragments

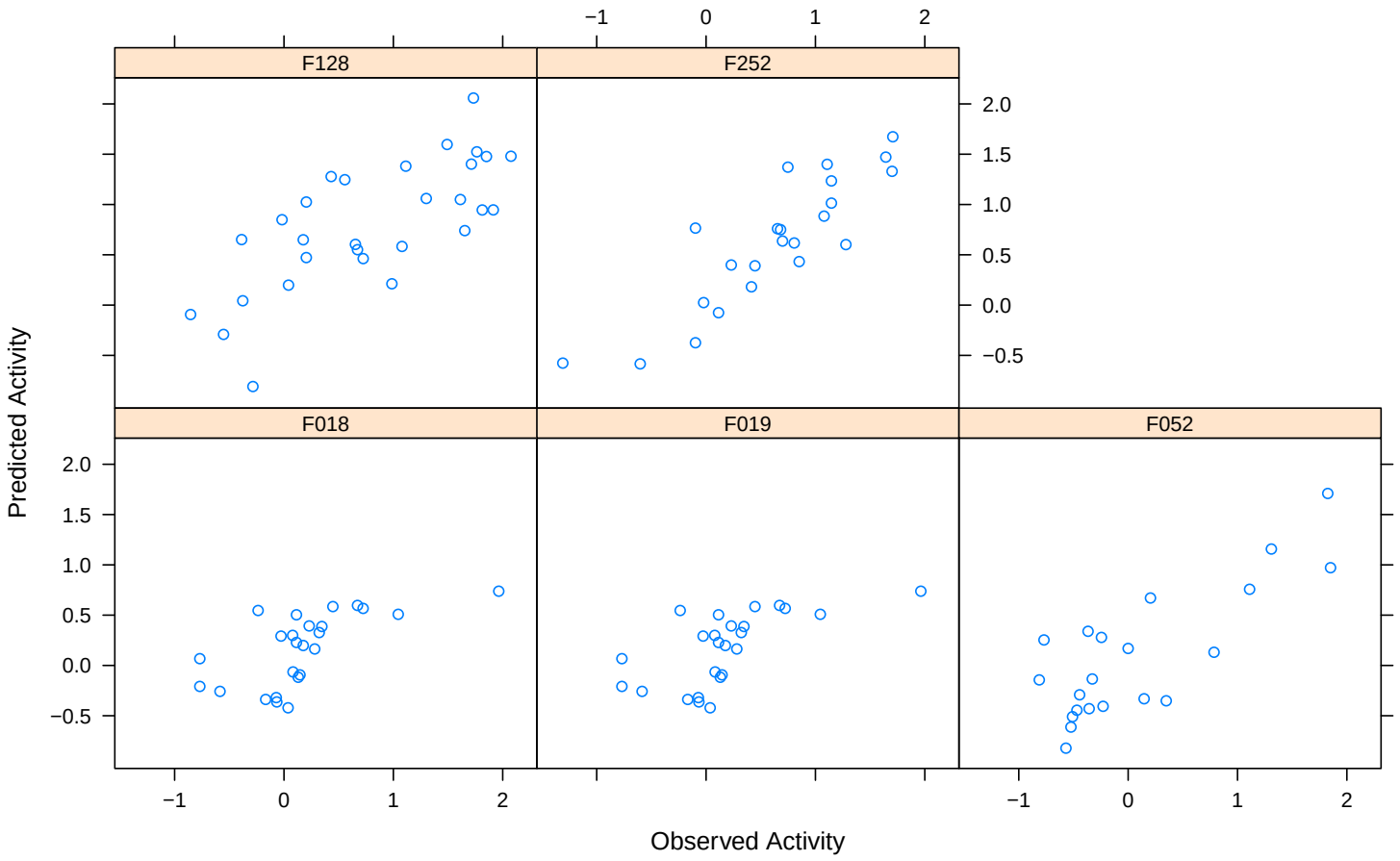
- Look at frequency of occurence of fragments
- Develop predictive models on fragment members, looking for local SAR
- *Pseudo-cluster* a dataset based on fragments
- Compound selection based on fragment membership

```
frag.counts <- by(frags, frags$frag, nrow)
barplot(frag.counts)
```

Characterizing fragment SAR

- Consider some of the larger series and see if there is an SAR within each of them
- Strategy
  - Identify fragments with 20 to 30 members
  - Merge the fragment data with descriptor data for the associated molecules
  - For each fragment series, build an OLS model using exhaustive feature selection
- I've precalculated a set of descriptors for this purpose (`d` in `data/frags.Rda`) or you can generate fragments yourself

Characterizing fragment SAR

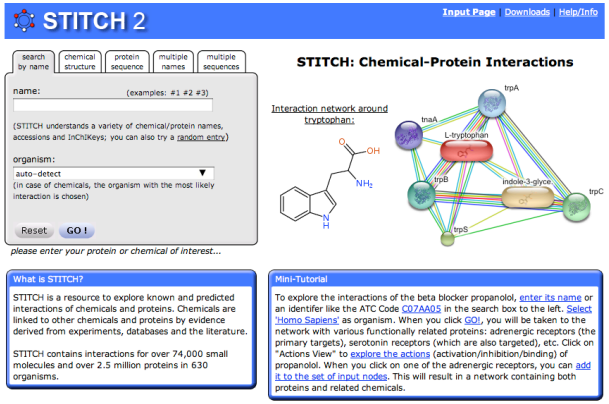


Predicted versus observed activity for 5 fragment series, based on OLS models and exhaustive feature selection

Part IV

Cheminformatics & R - `rpubchem`

Public chemical and biological data



DrugBank

ChemSpider
Building community for chemists

PubChem

eMolecules
Searching the World's Chemistry

ChEBI

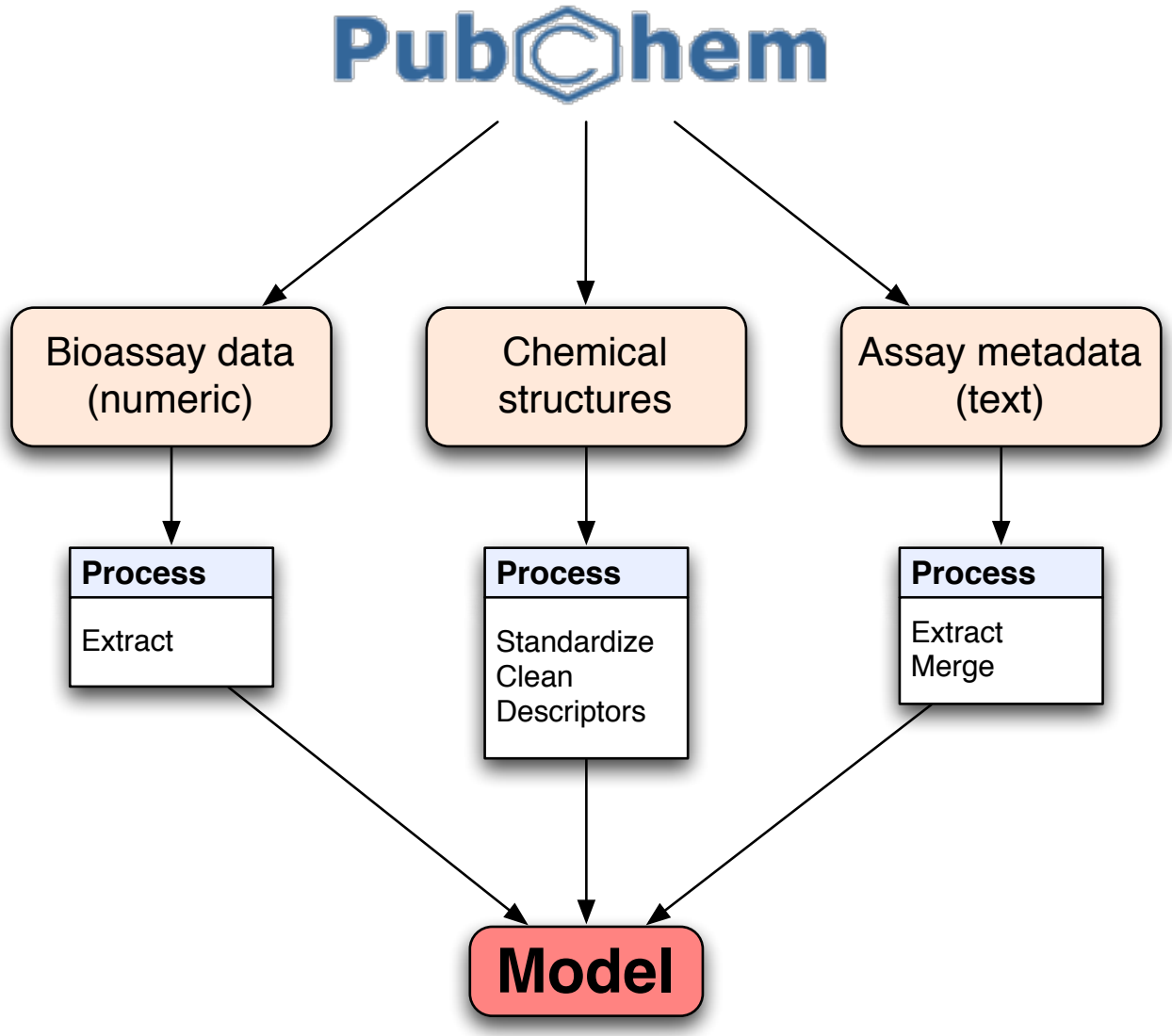
ChemBank

PDSP
K_i Database

BindingDB

chEMBL

Working with PubChem data



Working with PubChem data in R

- Access PubChem datasets using idiomatic R
- Minimize extra work (merge data, add metadata etc)
- End up with a `data.frame`, ready for modeling
- The usual sequence of tasks would involve
  - Get the assay data
  - Get structure data associated with the assay
  - Proceed with modeling

Working with PubChem data in R

- Access PubChem datasets using idiomatic R
- Minimize extra work (merge data, add metadata etc)
- End up with a `data.frame`, ready for modeling
- The usual sequence of tasks would involve
  - Get the assay data
  - Get structure data associated with the assay
  - Proceed with modeling

Getting assay data

- The workhorse method is `get.assay` which takes the AID as argument

```
assay <- get.assay(2018, quiet=FALSE)
```

- Returns a `data.frame` derived from the assay data CSV file and the assay description XML file form PubChem
- The *description*, *comments* and field descriptions and types are available as attributes of the `data.frame`
- **Note** - no structures are included at this point

Other ways to get assay data

- If you're looking for a group of assays, say related to HDAC inhibitors, perform a keyword search

```
## gives a vector of integer AIDs
aids <- find.assay.id("HDAC", quiet=FALSE)

## what are the assays?
sapply(aids[1:10], get.assay.desc)
```

Other ways to get assay data

- Find assays that a compound is active in

```
cid <- 68368
aids <- get.aid.by.cid(cid, type="active")
```

- Gives a vector AIDs that the compound is active in
- If you specify `type="raw"`, a summary is returned

Getting structure information

- Get structure data by CID or SID

```
cids <- get.cid(assay$PUBCHEM.CID[1:10])
```

- Gives you a `data.frame` containing name, structure and some properties

```
> str(structs)
'data.frame': 10 obs. of 11 variables:
 $ CID : chr "644436" "645300" "645983" "648874" ...
 $ IUPACName : chr "2-(2-methylphenyl)-1,3-benzoxazol-5-amine"
 $ CanonicalSmile : chr "CC1=CC=CC=C1C2=NC3=C(O2)C=CC(=C3)N"
 $ MolecularFormula : chr "C14H12N2O" "C18H15N3O2" "C24H34N6O3"
 $ MolecularWeight : num 224 305 455 407 290 ...
 $ TotalFormalCharge : int 0 0 0 0 0 0 0 0 0 0
 $ XLogP : num 3.1 3.2 3 4.5 2.1 2.9 1.8 2.5 0.7 1.7
 $ HydrogenBondDonorCount : int 1 0 1 1 2 2 1 1 2 2
 $ HydrogenBondAcceptorCount: int 3 4 7 5 4 3 7 8 5 4
 $ HeavyAtomCount : int 17 23 33 29 21 26 33 31 20 21
 $ TPSA : num 52 57 94.4 68 84.5 70.7 94.4 125 104 69.6
```

Ready to model

- Having gotten the `data.frame` of structures we can parse the SMILES into CDK objects and evaluate descriptors
- However, `get.cid` (and `get.sid`) will not work if you try and retrieve too many at one go
- The current design makes an Entrez URL, which can become too large and so is rejected
- Will be switching to the use of PUG
- In the meantime, retreive structure data in chunks

Getting CID data in chunks


```
## will likely fail
structs <- get.cid(assay$PUBCHEM.CID)

## better way, 300 CIDs at a time
library(itertools)
chunks <- as.list(ichunk(assay$PUBCHEM.CID, 300))
structs <- lapply(chunks, get.cid)
structs <- do.call("rbind", structs)
```

Part V

Extending `rcdk`

Why extend?

- `rcdk` doesn't cover the whole CDK API
- I add functions as and when I need them (or if someone asks)
- If you want to access CDK classes and methods, you can use `rJava` (such as `.jnew` and `.jcall`)
  - This can get a bit cumbersome
  - Use the soure code for inspiration
- For more complex stuff, you can write Java code that gets installed along with `rcdk`
  - Ideally this would get contributed back and incorporated into the `rcdk.jar` file that is bundled with the `rcdk` package

Structure of the `rcdk` package

Name	Date Modified	Size	Kind
data	May 7, 2010, 2:31 PM	--	Folder
rcdk	Today, 3:55 PM	--	Folder
data	May 3, 2010, 10:33 AM	--	Folder
DESCRIPTION	Today, 9:41 AM	4 KB	Plain text
inst	May 3, 2010, 10:33 AM	--	Folder
CITATION	May 3, 2010, 10:33 AM	4 KB	Plain text
cont	May 6, 2010, 8:24 AM	--	Folder
doc	Today, 2:47 PM	--	Folder
unitTests	Today, 9:41 AM	--	Folder
man	Today, 9:41 AM	--	Folder
NEWS	May 3, 2010, 10:33 AM	4 KB	Plain text
R	Today, 9:41 AM	--	Folder
tests	May 3, 2010, 10:33 AM	--	Folder
doRUnit.R	May 3, 2010, 10:33 AM	4 KB	R Code File

- If you add new functionality, good idea to add unit tests (using `RUnit`) under `rcdk/inst/unitTests/`

Structure of the cdkr project

Name	Date Modified	Size	Kind
data	May 7, 2010, 2:31 PM	--	Folder
rcdk	Today, 9:41 AM	--	Folder
data	May 3, 2010, 10:33 AM	--	Folder
DESCRIPTION	Today, 9:41 AM	4 KB	Plain text
inst	May 3, 2010, 10:33 AM	--	Folder
man	Today, 9:41 AM	--	Folder
NEWS	May 3, 2010, 10:33 AM	4 KB	Plain text
R	Today, 9:41 AM	--	Folder
tests	May 3, 2010, 10:33 AM	--	Folder
rcdkjar	Today, 3:49 PM	--	Folder
build.properties	May 3, 2010, 10:33 AM	Zero KB	Document
build.xml	May 3, 2010, 10:33 AM	4 KB	Text ...ument
jars	May 6, 2010, 8:24 AM	--	Folder
rcdk.iml	May 6, 2010, 8:24 AM	4 KB	Document
rcdk.ipr	May 6, 2010, 8:24 AM	20 KB	IntelliJ...ct File
rcdk.iws	May 6, 2010, 8:24 AM	40 KB	Document
src	May 3, 2010, 10:33 AM	--	Folder
rcdklibs	May 6, 2010, 8:24 AM	--	Folder
README.md	May 6, 2010, 8:24 AM	4 KB	Document

- Within `rcdkjar/`, run `ant jar` to build the jar file and copy into the `rcdk` package directory
- Generally, no need to touch `rcdklibs/`

Writing Java in R

- Don't like it, but sometimes it's easier than firing up the Java IDE. Need to be familiar with the CDK API
- Three main functions
  - `.jnew`
  - `.jcall`
  - `.jcast`

Writing Java in R

- First, lets look at a better way to get the atom count of a molecule
- Right now, we get a molecule and get the list of atom objects and return the length of the list
- But `IAtomContainer` has a method `getAtomCount()`

```
m <- parse.smiles("CCC")
.jcall(mol, "I", "getAtomCount")
```

- For static methods, the first cargument can be the fully qualified class name
- You do need to specify the return type in JNI notation

Writing Java in R

JNI Signature	Java Type
I	int
Z	boolean
B	byte
C	char
L*fully-qualified-class*;	fully-qualified-class
[*type*	*type*[]

- For the *L* signature, remember the semi-colon at the end

Writing Java in R

- Another example is getting the largest connected component
- We make use of the `ConnectivityChecker` class and the `partitionIntoMolecules` method

```
mol <- parse.smiles("CCC(=O)C[O-].[Na+]")
molSet <- .jcall("org.openscience.cdk.graph.ConnectivityChecker",
                 "Lorg/openscience/cdk/interfaces/IMoleculeSet;",
                 "partitionIntoMolecules", mol)
ncomp <- .jcall(molSet, "I", "getMoleculeCount")
```

Writing Java in Java

- When writing Java in R gets too tedious, switch to Java in Java
- You could write classes to do the work and then package them into a jar
- Copy the jar into the `inst/cont` of the `rcdk` directory
- Edit `.First.lib` in `R/rcdk.R` to add this jar to the classpath
- Alternatively edit the Java sources for the `rcdk` package
  - Need to access the Github repository

Future developments

- Update `rpubchem` to use PUG throughout
- Data table and depictions
- Streamline I/O and molecule configuration
- Add more atom and bond level operations
- Convert from `jobjRef` to S4 objects and vice versa
  - Would allow for serialization of CDK data classes
  - Is it worth the effort?
- Your suggestions